FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA

A PROCESS-BASED CONTROL FOR EVOLVABLE PRODUCTION SYSTEMS

Por:

José Nuno Palma e Belo

Dissertação apresentada na Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa para a obtenção do grau
de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Professor Doutor José António Barata de Oliveira

Júri:
Presidente: Professor Doutor Ricardo Luís Rosa Jardim Gonçalves
Vogal: Professor Doutor José António Barata de Oliveira
Vogal: Professor Doutor Pedro Alexandre da Costa Sousa

Lisboa
Junho 2011

A Process-based Control for Evolvable Production Systems

To my family
and friends.

Acknowledgements

It was a long journey since 2002, with its hardships and its ups and downs, which culminated in the work presented in this document. Through all the years spent studying Electrical and Computers Engineering, I was presented with many good moments and many new friends.

I would like to thank firstly to Professor José Barata whose help and guidance throughout the implementation of this thesis proved priceless. His faith in me and in my work were crucial in the path I chose for this research.

For many months I worked together with Nuno Pereira who gave valuable ideas and and tolerated my stubbornness in certain subjects. Also, Regina Frei and Giovanna Di Marzo Serugendo from Birkbeck College who counselled me in during the conception phase of this work.

All my friends, who listened my endless chatter about my doubts and problems related to this thesis and even helped me with some ideas, also deserve my deepest thanks. Specially, Eduardo Ortigueira, Frederico Gonçalves, Alexandre Rodrigues and Rita Otero.

Last but not least, I would like to thank my family, for providing me the ever necessary stability at home and economic support for the duration of my studies. Especially my mother, Maria José Rafael, and my aunt, Maria Luthgarda, who helped me in the correction of this document. Also, my girlfriend, Inês Silva, who knew in first-hand my deep most fears and anguishes during my hardest moments and always soothed me like beauty soothes the beast.

Resumo

Hoje em dia, as empresas num ambiente competitivo são obrigadas a adaptar-se às mudanças repentinas na indústria de manufactura. Há uma maior busca de novos meios para criar produtos com ciclos de vida curtos e a baixo custo, enquanto se mantêm os mesmos níveis de produtividade e qualidade. Isto gerou a necessidade de criar sistemas de manufactura cada vez mais ágeis, que se adaptassem facilmente e a baixo custo às mudanças no mercado.

Avanços nas tecnologias de informação permitem alcançar novos níveis de agilidade em sistemas de manufactura, abrindo portas para novas abordagens. Estes mesmos avanços ajudaram empresas em vários sectores, para além da manufactura, a aumentar a sua eficácia, sincronizando os processos dos seus vários departamentos com o uso de ferramentas de Gestão de Processos de Negócio.

Esta dissertação propõe um sistema que reage e se adapta a diferentes ordens de produção através de reconfiguração. Para alcançar esse objectivo, foi usado o conceito de Gestão de Processos de Negócio. Este conceito, já usado em muitas empresas, permite a que estas modelem o seu funcionamento interno de acordo com processos que podem ser alterados conforme as suas necessidades. Um sistema de manufactura que o use ficará igualmente ágil e ainda poderá alterar o seu funcionamento em concordância com as necessidades de outros departamentos da mesma empresa.

Para criar o sistema apresentado nesta dissertação foi usada uma arquitectura de multi-agentes, baseada em execução de processos. Cada agente contém uma base de conhecimento, usada pelos seus processos, que guarda informação interna ou externa. Este sistema pode ser usado, não só na área da manufactura mas também em qualquer outra área de uma empresa.

Esta dissertação apresenta também uma aplicação para o sistema na área da manufactura, baseada no conceito de Sistemas de Produção Evolutivos, no qual cada agente representa um recurso de manufactura que oferece serviços úteis para o processo de produção. Os

recursos, através dos agentes, podem agregar-se entre si para executar serviços em conjunto.
Palavras-chave: Sistema de manufactura, sistema multi-agente, ontologia, processo, BPM, EPS.

Abstract

Nowadays, companies in a challenging environment are compelled to adapt to the rapid changes in the manufacturing business. The search for new processes to create products with short life-cycles at low cost, while keeping the same levels of productivity and quality is greater than ever. This has generated the need to create even more agile manufacturing systems, which could easily adapt to the market changes at a low cost.

Advances in information technologies have allowed manufacturing systems to achieve new levels of agility, opening the doors to new approaches. These same advances helped companies in several sectors other than manufacturing to gain effectiveness through the synchronization of the processes of their several departments by using Business Process Management tools.

This thesis proposes a system that reacts and adapts itself to different production orders by means of reconfiguration. To reach this goal, the concept of Business Process Management was used. This concept, already used in many companies, allows them to model their inner behaviours with processes that can be changed according to their needs. A manufacturing system using this may become equally agile and alter its functioning in accordance with the needs of other departments of the same company.

To create the system presented in this thesis it was used a multi-agent architecture based on process execution. Each agent contains a knowledge base, used by its processes, that stores internal or external information. This system may be used not only in the manufacturing shop floor, but also in any other areas within a company.

This thesis also presents an application of the system to the shop floor, based on the Evolvable Production Systems concept, in which each agent represents a manufacturing resource that offers a given set of services useful to the production process. The resources, by means of the agents, may aggregate among themselves to execute services together.

Keywords: Manufacturing system, multi-agent system, ontology, process, BPM, EPS.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACC Agent Communication Channel

ACL Agent Communication Language

AEI Advanced Enabling Interface

AI Artificial Intelligence

AID Agent Identifier

AMS Agent Management System

API Application Programming Interface

BDI Belief-Desire-Intention

BPM Business Process Management

BPMS Business Process Management System

BPR Business Process Reengineering

CIM Computer Integrated Manufacturing

CPU Computer Processor Unit

DF Directory Facilitator

DL Description Logic

EAS Evolvable Assembly Systems

EPS Evolvable Production Systems

EUPASS Evolvable Ultraprecision Assembly Systems

FIPA Foundation for Intelligent Physical Agents

FMS Flexible Manufacturing Systems

GA	Genetic Algorithm
GUI	Graphical User Interface
HMS	Holonic Manufacturing Systems
IT	Information Technology
JADE	Java Agent Development Framework
KB	Knowledge Base
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
KSE	Knowledge Sharing Effort
MAS	Multi-Agent Systems
PLM	Product Lifecycle Management
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RMS	Reconfigurable Manufacturing Systems
SL	Semantic Language
SME	Small and Medium Enterprises
SOA	Service-Oriented Architecture
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
UML	Unified Modelling Language
URI	Uniform Resource Identifier
VE	Virtual Enterprise
WADE	Workflow Agents Development Environment
WfM	Workflow Management

WfMC Workflow Management Coalition

XML eXtensible Markup Language

XPDL XML Process Definition Language

Chapter 1

Introduction

When faced with unexpected faults, last minute order changes or totally different product order needs, manufacturing companies require continuous reconfigurations and adaptations in their shop floors [Barata, 2005]. This may be a problem in terms of costs and time spent, so a solution is needed in order to provide an advantage to manufacturing companies. So, to keep competitive, companies aim to improve their flexibility and agility while maintaining their productivity and quality [Leitão and Restivo, 2006].

The terms *flexibility* and *agility* are used separately, since they mean separate concepts: A flexible company is one that can adapt itself to produce a certain range of products efficiently, which means the products must be known prior to the system design or must not be very different from each other, while an agile company operates efficiently in a dynamic and uncertain environment. There is a considerable amount of research about manufacturing in the area of flexibility [Gullander, 1999, Vos, 2001, ElMaraghy, 2005] and also in the area of agility [Huff and Edwards, 1999, Leitao, 2004].

Currently, tendency goes to research systems that attain agility by using self-organisation and self-adaptation so that the response to changes in the manufacturing needs may be achieved with high productivity and low costs. Distributed Systems like Multi-Agent Systems (MAS) provide a solution for this situation, distributing the control of the system to a number of autonomous entities, reducing the complexity of the entire system by dividing it by each individual, assigning them well-defined tasks and responsibilities. This increases the flexibility and enhances fault tolerance [Merdan et al., 2008]. Emergent control approaches that can be implemented under the support of these concepts are agent-based manufacturing, Holonic Manufacturing Systems (HMS), Reconfigurable Manufacturing Systems (RMS), Evolvable Production Systems (EPS) and Evolvable Production Systems (EPS) [Ribeiro et al., 2008].

1.1 Objectives

The work described in this thesis falls under the domain of the EPS paradigm, which gets inspiration from areas like complexity theory, artificial life, autonomic computing, agents and self-organisation [Barata et al., 2007]. All of these areas affected, in one point or another, the objectives defined for this work. It was given more emphasis to the areas of agents and self-organization, since they were a central part of the thesis here presented, although the other areas also influenced some of the decisions during the planning.

The system described in this document aims to be an agile system with some self-organization capabilities, able to control shop floor resources in a dynamic way, so that when sudden changes in production orders or reorganization of the physical placement of the machines take place, it is able to adapt itself or be easily reconfigured without the need to be reprogrammed. In order to attain the desired level of flexibility and agility, the proposed system is set to fulfil the following requirements:

- **Modularity** - The modularization of manufacturing components assigns specific tasks to specific modules making them specialized and the system scalable. The combined efforts of two or more modules can result in a more complex behaviour. One analogy to this kind of reasoning is the human brain, where certain areas are specialized in a certain task and the combined effort of such tasks can result in more complex human actions, for example, eye coordination and several muscles coordination areas can combine in order to externalize the emotion of sadness [Damásio, 2003]. As stated before, this reduces the overall programming complexity of the entire architecture.

- **Reduce programming effort** - The act of reprogramming entire systems in order to cope with production changes or the addition or removal of a manufacturing component can be quite time-intensive and costly. So, one of the main objectives is to have a system with minimal or no reprogramming needs, switching all the work to configuration, which is easier and much faster.

- **Re-usability** - Manufacturing modules should be reused for as long as possible [Barata, 2005]. This implies the need to support updates and reconfiguration of their behaviours, thus adapting to new scenarios.

- **Self-Organization** - Modularized components have to be able to organize themselves in order to achieve certain goals, which may require the execution of complex tasks involving multiple modules. Also, modules have to respect each one's restrictions and needs as best as possible. This amounts to a self-organized system in which each constituent has an active part in it by exchanging information and reasoning over it.

In the approach chosen, the objective was the development of a process-based system, taking examples from the area of Business Process Management (BPM) which addresses many problems similar to the ones already cited, like the need to adapt to change in demands or the requirement for shorter life-cycles [Ryan K.L. Ko, 2009]. The system planned for this thesis is a Multi-Agent Systems (MAS) where each agent is able to execute processes[1] containing a certain set of activities. This approach would allow the reduction of programming effort and encourage re-usability.

Each constituent of the system (conveyors, cranes, manufacturing cells, etc.), represented by an agent, becomes a module with a certain degree of reasoning. This decentralized approach allows a modularization of the system and self-organization. Also, this modular point of view provides emergent properties to the system, that is, properties that cannot be predicted by analysing each part of the system separately. The smaller the parts and the less work entrusted to them, and therefore the higher granularity, the easier it is to coordinate and structure the system or change it altogether, providing it with a high degree of agility [Maraldo et al., 2006].

When developing Evolvable Production Systems (EPS), the highly dynamic life-cycle is the main problem to be considered, which, of course, contains some other problems itself, like the creation and re-engineering of systems (creation, dissolution and changing of a given production cell), the development of an architecture for an individual module and the development of an architecture that supports a society of modules [Onori et al., 2005]. These problems were addressed by the author as best as possible in the defined architecture.

[1]In the context of this work, processes and agent behaviours mean the same thing, as behaviours were programmed in a process-based way. When justifiable, these concepts are mentioned independently. In Chapter 3.3 a better explanation of BPM and processes is given, while in Chapter 5, the actual application of processes in this work is discussed.

1.2 Thesis Outline

This thesis is divided in seven chapters: *Introduction*, *Manufacturing Systems: A State-of-the-Art*, *Supporting Concepts and Technologies*, *A BPM-Based Architecture for EPS*, *A Process Model Specification*, *Implementation of the System* and *Conclusions and Future Work*.

The current chapter gives a brief introduction to the research problem, states the objectives outlined for this work and identifies some of the most important concepts used in this thesis.

Chapter 2 gives an overview of the state-of-the-art in manufacturing systems along with the current state of research in this area. Flexible and agile manufacturing systems are described in this chapter, as well as Evolvable Production Systems (EPS), which is the basis of this work.

Chapter 3 introduces the concepts used in the implementation of the work here described. Agents, ontologies and Business Process Management (BPM) are explained there. This chapter also gives an overview of the technologies used in the actual implementation of this thesis.

Chapter 4 is one of the main chapters in this thesis by presenting the generic multi-agent architecture that works under the BPM paradigm. In that chapter it is also explained the application of said architecture to the manufacturing environment, by detailing a higher-level EPS architecture.

Chapter 5 describes the theory behind the actual execution of processes, by detailing the model created for this effect. This is the most important chapter in this thesis, since all the work done is based on the technology in it described.

Chapter 6 describes the actual implementation of the architecture defined in Chapter 4 as well the implementation of the process model defined in Chapter 5. Later, it details how the architecture was configured to create the higher-level EPS architecture, also presented in Chapter 4. Lastly, the testing scenarios used in this work, as well as some results, are also presented.

Chapter 7 discusses the conclusions and contributions of this work. Also, it proposes more topics for further research.

Chapter 2

Manufacturing Systems: A State-of-the-Art

Nowadays, the needs are for highly customized products, with a short life-cycle, high quality and low costs. So, and with the increase of competitiveness, companies constantly need to achieve higher productivity, flexibility and agility in order to stay in the market.

Companies that can not solve their problems internally, tend to look for cooperation amongst themselves in order to increase competitiveness by creating Virtual Enterprises (VEs) or other types of alliances, fulfilling specific demands. This is the case of many Small and Medium Enterprisess (SMEs) that have poor engineering resources.

The use of competitive and up-to-date technologies is also one of the key factors for companies to stay on the market, and is a major topic in this thesis. With the creation of Computer Integrated Manufacturing (CIM), Flexible Manufacturing Systems (FMS) or different control architectures, new, cheaper and innovative ways of creating manufacturing systems were found gave companies a boost when competing with each other. Studies and works in this area also gave birth to concepts such as agile manufacturing or Evolvable Production Systems (EPS) and Evolvable Assembly Systems (EAS).

This chapter contextualizes the work in this thesis, introducing to manufacturing systems and giving an overview of the evolution in this area. It also describes new approaches that were studied to address some of the problems in the manufacturing business. This chapter also gives a glimpse of how manufacturing controls are an important part of this industry.

2.1 Manufacturing Systems

The chosen definition for the process of manufacturing may be found in [Groover, 2007]. According to this book:

> *Manufacturing can be defined as the application of physical and chemical processes to alter the geometry, properties and/or appearance of a given starting material to make parts or products; manufacturing also includes the joining of multiple parts to make assembled products. The processes that accomplish manufacturing involve a combination of machinery, tools, power and manual labour.*

Consequently, manufacturing systems, during the production process, involve machines, tools, material-handling systems and humans in charge of manual labour. The production process has also inputs of raw materials, information, energy, the guidelines that tell the system how to produce, product demands and external disturbances [Leitao, 2004]. This results in finished products, along with new information about performance or the current status of the system. Also, unused or useless material comes in the form of waste. An abstract model of a manufacturing system can be seen in Figure 2.1.

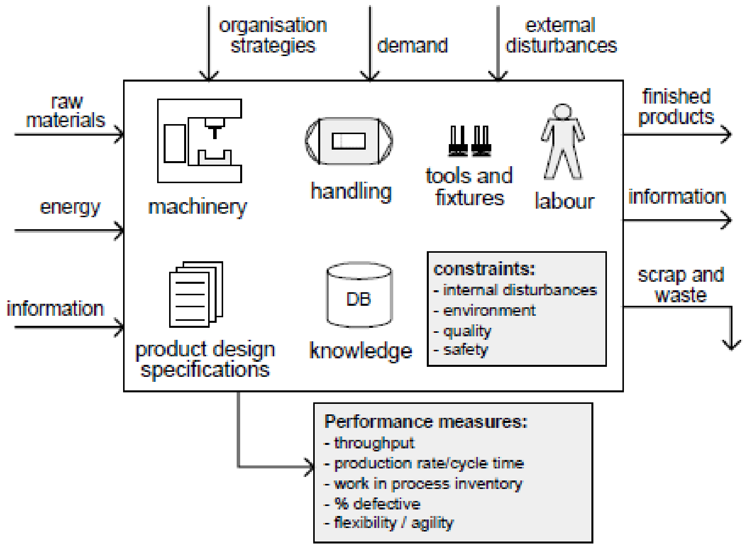


Figure 2.1: An Abstract model of a manufacturing system, found in [Leitao, 2004]. The system inputs, along with the resources contained in it, generate several outputs. The resources in the system have to obey to several constraints in order to work. There are also ways to measure the system performance.

To design manufacturing systems, all the factors shown in the previous figure need to be taken into account. Many of these factors determine how manufacturing control architectures are created. Figure 2.2 shows the steps required to design a control structure and where external factors influence that design.

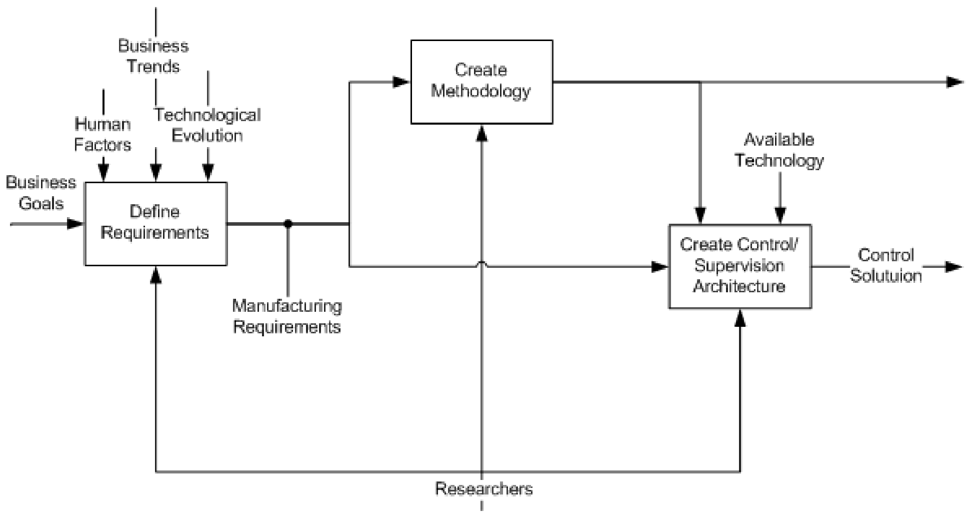


Figure 2.2: Activities to create a manufacturing control/supervision architecture, found in [Barata, 2005], which are the definition of requirements, methodology creation and architecture creation. Each activity depends on external factors.

Requirements, depending on several external factors, need to be defined in order to design the control structure. These requirements are applied to the creation of the methodology that will be used in the architectural planning of the control and supervision structure.

A planning like the one in Figure 2.2 may be executed several times, because the control structure defined at one time may not be suitable for future products [Barata, 2005]. Therefore, the manufacturing environment is in constant evolution. This evolution started at the end of the nineteenth century with the paradigm of *Craft Production*, where products were created to suit a single customer needs, using highly skilled workers and simple, but flexible tools [Piore and Sabel, 1984].

With the industrial revolution, the concept of *Mass Production* emerged: a product was manufactured in large scale using a rigid assembly line [Leitao, 2004]. This term was popularized by Henry Ford [Ford and Crowther, 1926, Gross et al., 1996].

After the late 1970s, studies were conducted to discover why the Japanese production techniques were more successful than the ones used in the western world. Such studies concluded that an efficient manufacturing technique was being used in Japan. This technique was later called *Lean Manufacturing* [Womack et al., 1990]. This concept relies on the principle of delivering high quality and low cost products with minimal waste. Waste is any activity that absorbs resources and does not add any value to the production [Barata, 2005].

With the constantly rising demand in the market for more personalized products at lower prices, while keeping the same quality levels, came the concept of *Mass Customization* [Pine and Davis, 1999]. This can be defined as a fast increase in the variety and customization of products while keeping the same costs. At its limit it is the mass production of individually customized products [Barata, 2005].

The integration of CIM in the manufacturing environment allowed new approaches to be designed. *Flexible Manufacturing* and *Agile Manufacturing* are two of such approaches that mark the latest years of research in the manufacturing domain. Since these concepts also fall in the domain of this thesis, they are described more thoroughly in Sections 2.2 and 2.3.

Another concept worth referring, is Business Process Reengineering (BPR), which is mainly an organizational philosophy applied in the higher management aspects of a manufacturing company. It has yet no application in the shop floor [Barata, 2005]. Ideally, a company should apply this concept in all its levels but this does not happen. BPR specifies how and when to redesign old processes, eliminating waste [Victor and Boynton, 1998]. Discussion on this subject will be picked up later in this document, since it is closely related to some of the concepts here used.

2.2 Flexible Manufacturing

With the advent of CIM, which introduced computerized control in the manufacturing environment [Browne et al., 1988, Camarinha-Matos et al., 1995, Miller and Walker, 1990], concepts like Flexible Manufacturing Systems (FMS) appeared to deal with the varying products and demands, in order to increase the competitiveness of companies throughout the manufacturing world by using the advantages of computers and automation. A FMS is composed of a reconfigurable set of work stations, interconnected by a flexible material handling system, and controlled by an integrated computational system [Upton, 1990].

A more specific definition may be found in the book *Flexible Manufacturing System* [Shivanand, 2006], where it is stated that:

> *A flexible manufacturing system (FMS) is an arrangement of machines [...] interconnected by a transport system. The transporter carries work to the machines on pallets or other interface units so that work-machine registration is accurate, rapid and automatic. A central computer controls both the machines and transport system.*

The adoption of the FMS approach meant that machines in a given system would be flexible enough to perform a wide range of tasks, which brought many advantages to the manufacturing domain, like increasing of productivity, decreasing of production costs, reduction of inventory and stocks and superior quality [Rembold et al., 1993, Ranky, 1990].

Even though machines were able to perform a certain range of tasks, they were never excellent performers performers at any of the individual tasks. This was one of the downsides of FMS. The systems would be flexible enough to cope with several different problems but would not resolve them in the best way possible. Another problem was that the FMS was flexible while producing a range of known products but, when it came to produce something from a different and previously unknown product family, it became inflexible [Leitao, 2004], because agility was not a concern at the time and the life cycle support lacked from the concept [Barata, 2005].

2.3 Agile Manufacturing

Nowadays, the unpredictability of the factors that influence the manufacturing domain, like the markets or society itself, caused for the research of a new and innovative way to handle systems. The solution for these problems came in the form of agile manufacturing, which was first mentioned in a report by Nagel and Dove [R. Nagel, 1992].

Agile manufacturing is a step forward to the previously mentioned FMS and similar systems. Even though a FMS can be used to produce a wide range of products and can accommodate some internal changes, they only work in a predictive environment. This is not the case if the system is agile, since agile manufacturing deals better with things that cannot be controlled [Maskell, 2001] or, in other words, uncertain environments.

Agility is used in many different areas of manufacturing, from the lower shop floor to the management of manufacturing companies. One may say that the concern for agility is a company-wide effort, where all the areas need to be integrated in order to obtain the best results. A successful implementation of an agile manufacturing system requires the following points [Barata, 2005]:

- **Political decisions** - Regulations are needed to help cooperation and innovation. Also, political decisions may affect the bounds in which a company should work.
- **Business cooperation** - Companies should be able do diversify cooperative relationships by creating virtual partnerships, in which they share their business competencies with each other, providing focused services and products to the customer.
- **Customer focus** - It is important to create a philosophy to focus the company on the customer. Creating solutions to add value to products or services is vital for companies to gain the attention of the customer, enabling more demand and higher profit.
- **Information technology** - In all areas of an agile company, computational support is essential, from creating a virtual partnership with another company to controlling the shop floor.
- **Processes re-engineering** - Involves identifying what processes must exist and redesigning them if needed. Process-centric approaches in companies are widely used nowadays. Since this is one of the main subjects of this thesis, this concept will be further discussed in Section 3.3.
- **New work organization** - An organization based on teams and cooperation, with skilled and autonomous workers must be implemented. These workers need to be highly trained, since their competences may fall under several domains within the companies and autonomy is highly needed.
- **New agile shop floor strategies** - As stated before, the current approaches, like FMS, do not deal well with uncertain environments. A new strategy is needed to cope with these problems, like EPS. Also, a careful study of all the entities involved in the process of manufacturing is needed in order to create a new methodology that integrates them.
- **Willingness to change** - All the actors involved must be constantly monitoring the surrounding environment and be willing and prepared to react in case of need.

2.4 Evolvable Production Systems

In recent years, a new concept of manufacturing systems was created, even though it has yet no actual implementation in the real manufacturing world: the Evolvable Production Systems (EPS)[1]. Much research on this subject was already accomplished by many different authors [Frei and Barata, 2008, Barata and Onori, 2006, Shen et al., 2006, Maraldo et al., 2006].

An EPS is a system that can dynamically adapt itself to new products and production scenarios. This means that the addition and removal of manufacturing modules and changes in the production orders stimulate the system to adapt to new scenarios at run-time, without the need to completely stop for reprogramming. This can be achieved by designing systems that can integrate any form or type of equipment, which, in turn, must be broken down into smaller, process oriented components. Ontologies and Knowledge Bases (KBs) need to be created to structure the assembly process. Once the process requirements are captured, an assembly platform will be attainable. This platform is linked to the product designers, in order for them to know the system capabilities and the constraints related to the product design. This is a highly adaptable and re-configurable system [Maraldo et al., 2006].

The EPS modular point of view provides emergent properties to the system, which are properties that cannot be predicted by analysing each part of the system separately. The smaller the parts and the less work entrusted to them, the easier it is to coordinate and structure the system [Shen et al., 2006, Maraldo et al., 2006].

Studies concluded that EPS needed a certain set of qualitative features to be described, as can be seen below [Shen et al., 2006, Barata and Onori, 2006]:

- **Module** - Represents any unit that can process an operation and integrates a specific interface. A module may represent a single manufacturing component, like a driller, or represent a coalition of several components, like a cell.
- **Granularity** - The lowest level of device considered within a reference architecture. For instance, if a robotic arm and a gripper are considered individually, they may communicate and new characteristics may arise, like flipping an object. The lower the level, the higher the emergence.

[1]In this work, most of the references presented for EPS call it Evolvable Assembly Systems (EAS). These concepts share the same meaning and only have different names. This might be due to different researches about the same subject being performed at the same time.

- **Plugability** - The ability to add or remove system components. This aspect is very important when considering a system that may need new manufacturing resources when a new product order is issued.
- **Reconfigurability** - The ability to rearrange available system components to perform new, but pre-defined, operations when a new module is added or to discard operations when an existing module is removed.
- **Evolvability** - If a fully *reconfigurable* system platform exhibits an emergent behaviour which introduces new or refined levels of functionality. This may be achieved by applying the previously stated points to a system.

In practice, in order to comply with the qualitative features above, an EPS containing a certain set of quantitative features to address each required quality has to be implemented. Table 2.1 adapted from [Shen et al., 2006, Barata and Onori, 2006] shows these features.

Table 2.1: EPS qualitative versus quantitative features.

EPS qualitative features	**EPS quantitative features**
Evolvability-conformity	Skills repository and Management
Plugability and Reconfigurability - control specifications	Module description/blueprint
Plugability-user requirements	Application guidelines
Granularity-Safety conformity	Safety certification procedures
Evolvability and Safety	Rules related to emergent behaviour
Plugability-practical implementation	EPS "wrapper" solution: hardware
Evolvability-practical implementation	EPS "wrapper" solution: software
Evolvability and Precision	EPS architecture approach (granularity to lowest level)

An EPS system requires a virtual repository to store all the structures needed for users to comply with specifications and also the reference architecture. Specifications must be added in order to attain plugability, so that external users can import their equipment to the system. Also, specifications are needed for data exchange and adaptation. Emergence may cause undesirable characteristics to appear and raise safety issues, so rules and safety certification procedures have to be enforced. Modules need to interact between each other and with the hardware, so wrapper interfaces, also called Advanced Enabling Interface (AEI), need to be created. This also implies that legacy components may be adapted to the EPS format. Finally an extremely well defined reference architecture has to be implemented.

The main building blocks of an EPS are the modules, which may represent physical components of the architecture or aggregations of these components that present the emergent properties resulting from such aggregation. Physical modules must describe the set of characteristics of the components they represent and should also capture the behaviour of the component and realize the necessary control actions that must be issued for the behaviour to be accomplished [Onori et al., 2005]. These behaviours are viewed by the system as skills and each skill represents the capability of the module to perform a certain task. A skill execution may involve performing a sequence of control actions offered by the component controller. The behaviours that emerge out of the interaction of the individual modules represent the complex functionalities, or complex skills, of the system [Onori et al., 2006]. These complex skills are offered by higher level modules representing the aggregation of other modules.

An example of an EPS is CoBasa [Barata, 2005]. Figure 2.3 shows the basic functioning of this system.

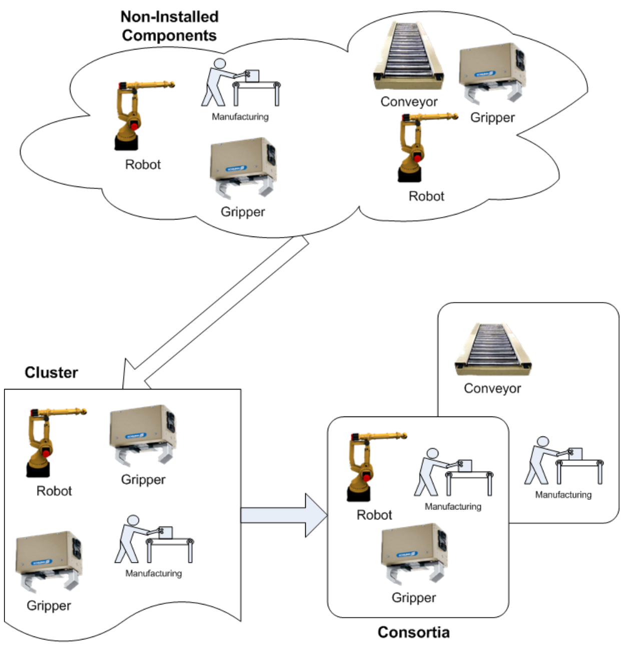
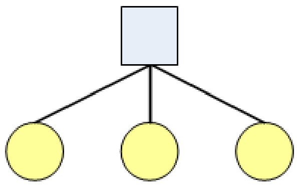


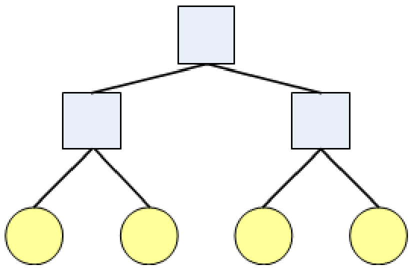
Figure 2.3: CoBasa consortia formation.

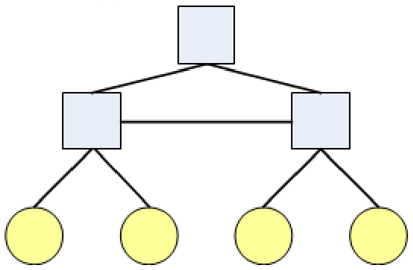
The CoBasa architecture is based on consortia formation, where each module is placed in a cluster waiting for a re-engineering opportunity. When this opportunity arises, several consortia are created between the modules, each with a specific operation plan that fits in the production.

2.5 Manufacturing Controls

One of the things that influences the final performance of manufacturing systems is their control architectures. Traditionally, the architectures can be classified as centralised, hierarchical, modified hierarchical and heterarchical [Dilts et al., 1991]. These architectures can be seen in Figure 2.4.


(a) Centralized.


(b) Hierarchical.


(c) Modified hierarchical.

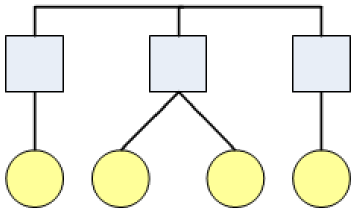

(d) Heterarchical.

Figure 2.4: The traditional control architectures: centralized, hierarchical, modified hierarchical and heterarchical.

The centralized architecture consists of one central node that coordinates all the inferior nodes. This approach provides simpler coordination issues, since most of the logic is placed inside the central node, but has several disadvantages, like the difficulty of modification or extension or the excessive complexity of the central node, in comparison to the inferior ones.

The hierarchical architecture releases the central node from most of the complexity by transferring it to inferior nodes. This can provide adaptive behaviours and can reach a nearly optimal performance under stable situations but has poor fault tolerance.

The modified hierarchical architecture derives from the hierarchical architecture by adding connections between the inferior nodes. This aims to improve disturbance responses and provides better expandability to the system.

The heterarchical architecture takes the central node out of the picture, distributing control over the inferior nodes. The advantage is the great flexibility of this type of control but, on the other hand, the absence of a central node can render impossible an appropriate control over the system.

2.6 Conclusions

Even though the technological evolution of manufacturing covers many concepts, some of them described in the previous sections, the work of this thesis was solely to create and study a new approach in the context EPS. And, even then, implementing a full EPS with all the described features would require an amount of work too great for a master thesis. This chapter was written only to contextualize the reader in the current state of manufacturing systems, giving the necessary background to this work.

As already stated, this work evolved around a process-based approach which eases the configuration of shop floor control. This does not mean that it is turned into a simple task, as configuring a full-fledged EPS would still require a great amount of labour and many other considerations beyond what was achieved through the course of this work. This thesis may be viewed as a contribution to an on-going study on new approaches for more agile and better manufacturing control architectures, able to cope with many of the problems described in this chapter.

Chapter 3

Supporting Concepts and Technologies

The development of this thesis required the support of some concepts and technologies. The motto *"Standing on the shoulders of giants"* was taken seriously and a great deal of effort was spent in studying different approaches and technologies to implement this work, culminating in the choice of the ones that actually support this framework.

The purpose of this chapter is to introduce the reader to these underlying concepts in order to have a better comprehension of the following chapters and of the reasons that influenced their choices. What is described in this chapter is just an introduction to the concepts and the references scattered throughout the text should be consulted for better insight on the subjects.

This chapter firstly explains the idea of Multi-Agent Systems (MAS) and the used technology for their implementation, Java Agent Development Framework (JADE). Secondly, the ontology concept is described. In the context of this thesis, the used ontology language was Web Ontology Language (OWL), which is based on Resource Description Framework (RDF). Also, for querying and asserting facts on ontological KBs, SPARQL Protocol and RDF Query Language (SPARQL) was used, along with its variant that allows the update of KBs. All of these languages and respective implementation technologies, Jena and Protégé, are also described in the ontology section. Lastly, an introduction to Business Process Management (BPM) is given, explaining what it is and why it is in the scope of this work.

3.1 Multi-agent Systems

The traditional manufacturing control systems do not support efficiently the current requirements imposed to the manufacturing systems [Leitão and Restivo, 2006]. Leitão also states that, because of the increase of powerful, inexpensive and widely available computational resources, the architectures evolved from centralised to distributed and dynamic approaches.

Multi-agent systems, being distributed systems by nature, tend to be used now in order to solve the flexibility problems in manufacturing systems. Many researches and studies have been focused in this area [Barata and Camarinha-Matos, 2002, Barata et al., 2005, Colombo et al., 2006].

3.1.1 Individual Agents

The definition of an agent is a controversial subject, as there is no consensus about what the exact definition is. Nonetheless, in this work, the adopted definition was [Wooldridge, 2002]:

> *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.*

This means agents need to reason about their surrounding environment in order to take an autonomous action. Even though this may mean the agents are intelligent, a simple program that gathers information, reasons over it and takes an action, like, for example, the software that detects the Computer Processor Unit (CPU) temperatures and fires an alarm when they are too high, can hardly be described as an intelligent program. Below, a list of the kinds of capabilities one might expect from an intelligent agent is presented [Wooldridge and Jennings, 1995]:

- **Reactivity** - The capability an agent has to react to environment changes in order to satisfy its objectives.
- **Pro-activeness** - The capability of agents to have a goal-directed behaviour in order to achieve their objectives.
- **Social ability** - The capability agents have of interacting with each other and to achieve their objectives.

Creating purely reactive or pro-active systems is not very difficult as they are made all the time, but a system that balances these characteristics is the key for an intelligent agent functioning. Also, the social ability is not only the passing of bit streams from an agent to another, but also their capability of negotiating and cooperating to achieve a common goal.

Programmers that use object-oriented programming may think that agents are more or less the same but there are differences [Wooldridge, 2002]:

- Agents have a stronger notion of autonomy than objects and they decide for themselves whether or not to perform a particular action on external request.
- Agents are capable of flexible behaviour, unlike objects.
- A multi-agent system is inherently multi-threaded, with each agent having at least one thread of control.

There may also exist some confusion between the concept of agency and expert systems but there are differences [Wooldridge, 2002]:

- 'Classic' expert systems are not coupled to any environment to act in. They rather act through a user as a 'middleman'.
- Expert systems are not generally capable of flexible behaviour (reactive, proactive and social).

3.1.2 Agent Typologies

Agents can be classified into three architectural types, according to their attitude towards the surrounding environment [Wooldridge and Jennings, 1994]: deliberative, reactive and hybrid.

3.1.2.1 Deliberative Agents

This type of architecture was first described in the book *Logical Foundations of Artificial Intelligence* [Genesereth and Nilsson, 1987] and is characterized by agents that reason over an internal representation of the environment surrounding the agent. This internal representation is what the agent believes the environment should be, which might not be the real environment These agents have a goal-oriented behaviour dictated by their actions.

The most well-known architecture inside the deliberative agent paradigm is the Belief-Desire-Intention (BDI) architecture, which contains data structures loosely corresponding to these mental states [Wooldridge, 2000]. The BDI architecture includes the agent knowledge about the environment (beliefs), preferred states to achieve in the long-term (desires) and planned decisions to be made to complete a plan (intentions).

3.1.2.2 Reactive Agents

Agents that fall inside the reactive paradigm do not contain an internal representation of the surrounding environment. They merely react to the environment without reasoning over it [Wooldridge, 2002]. The ideas behind this type of architecture are [Brooks, 1991b, Brooks, 1991a]:

- Intelligent behaviour can be generated without explicit representations of the kind that symbolic Artificial Intelligence (AI) proposes.
- Intelligent behaviour can be generated without explicit abstract reasoning of the kind that symbolic AI proposes.
- Intelligence is an emergent property of certain complex systems, where the interaction of several simple and reactive nodes may generate more complex behaviours.

3.1.2.3 Hybrid Agents

The hybrid architecture is an alternative to the limitations imposed by both the deliberative and reactive architectures. Purely reactive agents are not capable of implementing a goal-oriented behaviour, while deliberative agents may become incapable of rapidly responding to external stimuli.

This hybrid type of architecture merges the advantages of both the deliberative agents and the reactive agents. This is accomplished by creating an hierarchy of layers in the agents [Wooldridge, 2002]. Each one of these layers may represent a reactive or a pro-active component. There should be a minimum of two layers each representing a different typology. Control flow in layers can be horizontal or vertical. An example of horizontal layered architectures is the TouringMachine [Ferguson, 1992] and for the vertical layered architectures, the InteRRaP [Muller and Pischel, 1993].

3.1.3 Agent Communication

Since a single agent represents only a part of a multi-agent system, agents must communicate so that all the parts of the system may interact and achieve common goals. To attain this purpose, the agents need to understand each other via communication languages and ontologies. Since ontologies are one of the main subjects in this thesis, they are described in detail in Section 3.2.

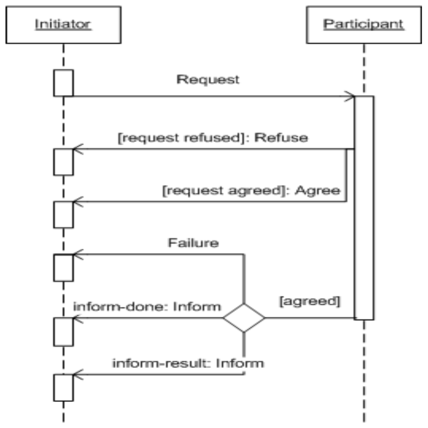
The Agent Communication Language (ACL) was directly influenced by the Speech Act theory, which treats communication as an action. One main ACL was created by the DARPA-funded Knowledge Sharing Effort (KSE): The Knowledge Query and Manipulation Language (KQML). This language was intended to be an envelope language for agent messages, in which the agent can state the intended use of the message. The contents of the messages did not matter for this language. KSE released the Knowledge Interchange Format (KIF) for the description of the contents of the messages.

Table 3.1: The FIPA parameters for ACL messages. Adapted from [FIPA, 2002a]

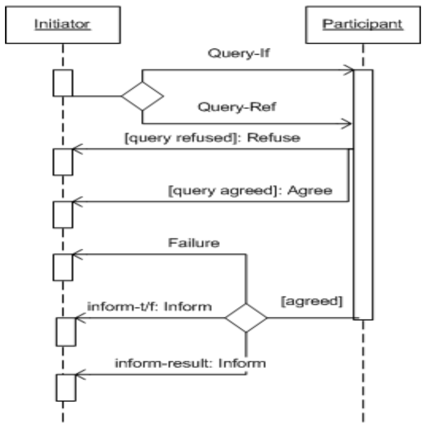
Parameter	Description
performative	Denotes the type of the communicative act of the ACL message.
sender	Denotes the identity of the sender of the message, that is, the name of the agent of the communicative act.
receiver	Denotes the identity of the intended recipients of the message.
content	Denotes the content of the message; equivalently denotes the object of the action. The meaning of the content of any ACL message is intended to be interpreted by the receiver of the message.
language	Denotes the language in which the content parameter is expressed.
ontology	Denotes the ontology(s) used to give a meaning to the symbols in the content expression.
protocol	Denotes the interaction protocol that the sending agent is employing with this ACL message.
conversation-id	Introduces an expression which is used to identify the ongoing sequence of communicative acts that together form a conversation.

Later, the Foundation for Intelligent Physical Agents (FIPA) released the FIPA-ACL language, which is almost identical to KQML. The FIPA-ACL describes every communicative act with both a narrative form and formal semantics based on modal logic, and it also includes a normative description of a set of high-level interaction protocols, such as requesting information and contract-net [Labrou et al., 1999]. This language was used throughout the implementation of this work, so it deserves a little more explanation. A message in this language contains a set of one or more message parameters. The most important parameters are defined in the Table 3.1.

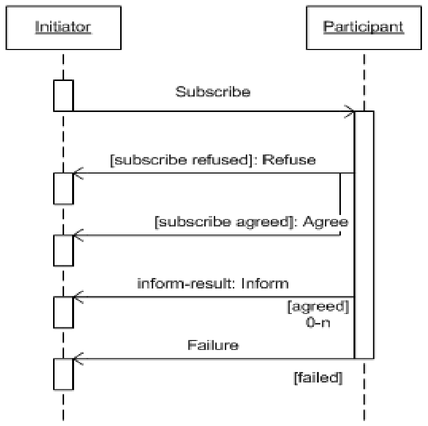
The table takes out one important feature of this type of messages: the ability to contain user-defined fields, which allows a user to add fields to a message that make sense in the scope of his work. Also, the *performative* is very important, which defines the communicative act attached to the message. Communicative acts define whether the message is a request, a proposal or any other type defined in the FIPA specifications [FIPA, 2000].



(a) The FIPA request protocol.



(b) The FIPA query protocol.



(c) The FIPA subscribe protocol.

Figure 3.1: Some of the communication protocols specified by FIPA for ACL messages.

FIPA-ACL supports a series of message exchange protocols, which use the communicative acts to differentiate the messages during the exchange. In a given protocol the *conversation-id* should always be the same in order for the agents to know that the messages belong to the same conversation. In the context of this work, only three protocols were considered, even though there are many more specified by FIPA. The *Request* protocol consists on an agent requesting another agent to perform a certain action, which may be successful or not [FIPA, 2002c]. The *Query* protocol is used when an agent needs to be queried for information at a given moment [FIPA, 2002b]. The *Subscribe* protocol allows an agent to request a receiving agent to perform an action on subscription and subsequently when the referenced object changes [FIPA, 2002d]. All of these protocols are shown in Figure 3.1.

3.1.4 JADE

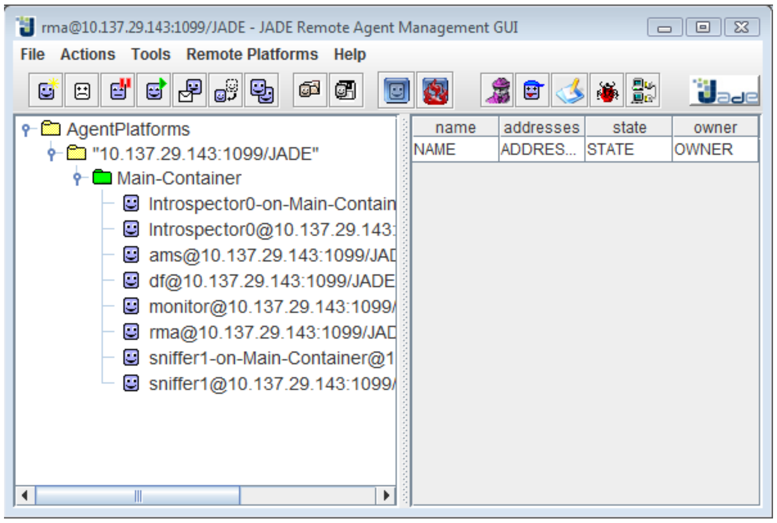
MAS can be implemented using regular programming languages, such as Java, by implementing the needed features for agent communication, ontology support, yellow and white pages service, message encoding, parsing and transport and agent life-cycle management services [Leitao, 2004]. Several platforms that support such features are already implemented and Java Agent Development Framework (JADE) is among them [JADE, 2010].

JADE is a framework for creating and managing agents that is compliant with the FIPA specifications. It provides a naming service, yellow-page service, message transport and parsing service, and a library of FIPA interaction protocols. Additionally, JADE provides the mandatory components defined by FIPA to manage the agent platform, which are the Agent Communication Channel (ACC), the Agent Management System (AMS), and the Directory Facilitator (DF). The DF is a particularly useful service, since it provides yellow pages services needed to find other agents and services in the network by sharing the information it contains when queried.

An agent in JADE must be able to carry out several concurrent tasks in response to different external events. In order to make agent management efficient, every JADE agent is composed of a single execution thread where all its tasks are modelled. These tasks can be implemented as what is called *behaviours*, which are simply atomic tasks that are executed sequentially inside the agent thread. A developer who wants to define an agent-specific task should define one or more behaviours and add them to the task list [Bellifemine et al., 2010].

The communications between agents are performed through ACL messages. JADE provides the FIPA Semantic Language (SL) content language and the agent management ontology, as well as the support for user-defined content languages and ontologies. Also, the FIPA-ACL message exchange protocols and message parameters, described in the previous section, are also supported by JADE. Some of the protocol implementations have simplified versions where less messages are exchanged for optimization purposes. In most of the cases, there is an omission of the section of the protocols where an acceptance or refusal is made.

JADE also provides a generic Graphical User Interface (GUI) that may be used to monitor active agents and communications between them, among other details. It can also be used to influence the system itself by creating and killing agents as well as performing other actions. This GUI is shown in Figure 3.2. The JADE GUI comes with several useful tools to gather specific information from the deployed agents, which include the sniffer agent, responsible for tracking all the messages sent in the system, and the introspector agent, which may inspect the states of the behaviours inside the agents.


rma@10.137.29.143:1099/JADE - JADE Remote Agent Management GUI
File
Actions
Tools
Remote Platforms
Help

- AgentPlatforms
  - "10.137.29.143:1099/JADE"
    - Main-Container
      - Introspector0-on-Main-Contain
      - Introspector0@10.137.29.143:
      - ams@10.137.29.143:1099/JAD
      - df@10.137.29.143:1099/JADE
      - monitor@10.137.29.143:1099/
      - rma@10.137.29.143:1099/JAD
      - sniffer1-on-Main-Container@1
      - sniffer1@10.137.29.143:1099/

name	addresses	state	owner
NAME	ADDRES...	STATE	OWNER

Figure 3.2: The JADE GUI main window, where the user may get essential information about the currently running agents.

Since the JADE framework is implemented in Java, it is possible to integrate it with other frameworks in the same programming language in order to extend the MASs to support new features. In the context of this thesis this has been done with the Jena framework, which is described in Section 3.2.4

3.2 Ontologies

In philosophical terms ontology is the science of the kinds and structures of objects, properties, events, processes and relations in every area of reality and it is often used by philosophers as a synonym of metaphysics [Smith, 2009]. Later, ontologies became popular in other areas outside philosophy, like knowledge management, artificial intelligence, cooperative information systems and other areas in which a common knowledge of things was needed.

An ontology provides a common understanding of concepts inside a particular domain. This opens the way for shareable and reusable KBs. Several works, which motivated the evolution of ontologies in this area, were already accomplished [Gruber et al., 1993, Guarino et al., 1993, Guarino and Poli, 1995].

In practical terms an ontology is a formal explicit description of concepts or classes in a certain domain, properties of each concept describing various features and attributes of the concept, and restrictions on properties [Noy et al., 2001]. Each concept can be instantiated in order to form an individual. Inference rules can also be defined in order to fine-grain the relationships between concepts, properties and restrictions. As an example, the concept *Person* and properties *name* and *sibling*, one could create two instances of *Person*, one whose value for the property *name* is *Alice* and the other is *Bob*. These two instances may then be connected using the property *sibling*. This would be the same as stating *The Person whose name is Alice is a sibling of the Person whose name is Bob.*

According to [Barata, 2005], the probable reasons why ontologies are becoming more and more important are:

- **Increasing use of computer agents** - Software agents acting independently from humans need to have a common language in order to understand each other.
- **More Knowledge Management Practices** - Organizations need to structure and maintain information. This makes a common information definition a very valuable asset.
- **The importance of the World Wide Web** - The growing importance of the web, not only for organizations but also for personal users, led to the creation of the concept of Semantic Web, which provides means for defining machine-understandable data as opposite to the current concept where all the data is only human-understandable.

Several ontology implementations are currently in use to aid the implementation of computer-based systems. Some of them are:

- **Knowledge Interchange Format (KIF)** - It is a variant of the language of the first-order predicate calculus, motivated by the goal of developing an expressive, flexible, computer- and human-readable medium for exchanging knowledge bases [Smith, 2009].
- **Ontolingua** - It is an extension of KIF with additional syntax, and organizes knowledge in object-centered hierarchies with inheritance.
- **DAML+OIL** - It is the ontology of the Defense Advanced Research Projects Agency, a combination of the DARPA Agent Markup Language, with the so-called Ontology Inference Layer. It was created to facilitate the concept of Semantic Web.
- **Web Ontology Language (OWL)** - The Web Ontology Language is an extension of the Resource Description Framework (RDF) and Resource Description Framework Schema (RDFS) specifications used to describe the classes and relations between them that are inherent in Web documents and applications. It can describe classes and properties in complex ways allowing even more expressiveness than RDF and RDFS.

In the manufacturing research domain, ontological descriptions have been already used in several multi-agent systems in order to achieve more flexible controls [Merdan et al., 2008, Mercian et al., 2006, Alsafi and Vyatkin, 2010, Pouchard et al., 2000].

3.2.1 RDF

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web based in eXtensible Markup Language (XML). It was created with the intention to be used to represent meta-data about Web resources, such as the title, author, and modification date of a Web page. However, by generalizing the concept of a *Web resource*, RDF also started to be used to represent information about things that can be identified on the Web, even when they can not be directly retrieved on it, allowing the creation of off-line descriptions.

RDF is intended for cases in which this information needs to be processed by applications, rather than being only displayed to people. RDF provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning. Since it is a common framework, application designers can leverage the availability of common RDF parsers and processing tools. The ability to exchange information between different applications means the information may be made available to applications other than those for which it was originally created.

RDF is based upon the idea of identifying things using Web identifiers, or Uniform Resource Identifiers (URIs), and describing resources in terms of simple properties and property values. This enables RDF to represent simple statements about resources as a graph of nodes and arcs representing the resources, and their properties and values.

The example graph in Figure 3.3 represents the statement *There is a resource of the type http://www.fct.unl.pt/ontologies/example.owl#Person, whose given identification is http://www.fct.unl.pt/ontologies/example.owl#alice, whose name is Alice and whose surname is Cooper.* Or, more simply put, *Alice Cooper is a Person.*

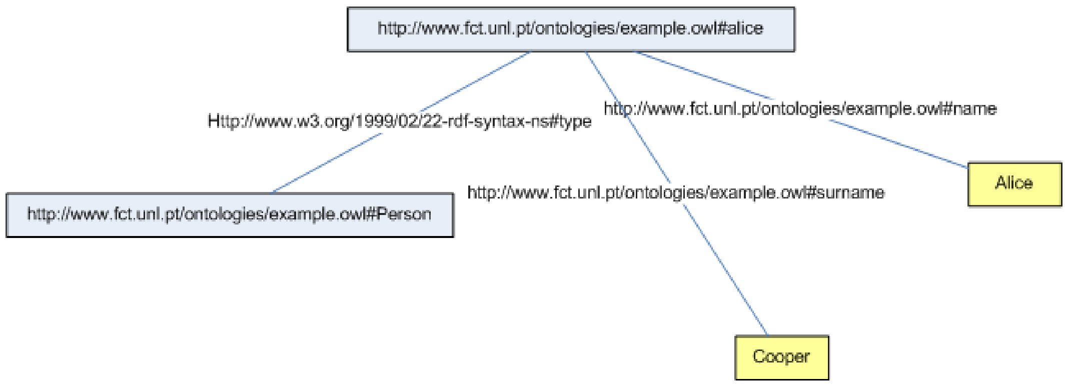

http://www.fct.unl.pt/ontologies/example.owl#alice
Http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.fct.unl.pt/ontologies/example.owl#name
http://www.fct.unl.pt/ontologies/example.owl#surname
http://www.fct.unl.pt/ontologies/example.owl#Person
Alice
Cooper

Figure 3.3: An example of a RDF graph. The resources are represented by the blue nodes, values are represented by the yellow nodes and properties are represented as arcs.

RDF has much more capabilities and, along with its integration with Resource Description Framework Schema (RDFS), provides an expressive language to create simple ontologies but not as expressive as OWL, which is explained in the next section. More details on these subjects can be found on the specifications web sites [W3C, 2004d, W3C, 2004c].

3.2.2 OWL

As described earlier, Web Ontology Language (OWL) is an ontology language for the Semantic Web built on top of the RDF language. As RDF, it is designed for applications that need to process the content of information instead of just presenting information to humans. OWL provides greater machine understanding of Web content than that supported by XML, RDF, and RDFS by providing additional vocabulary along with a formal semantics. OWL provides three sublanguages [W3C, 2004b]:

- **OWL Lite** - Allows functionality for the users that primarily need a classification hierarchy and simple constraints.
- **OWL DL** - Is more complex than OWL Lite by supporting maximum expressiveness while retaining computational completeness, meanings that all the conclusions are guaranteed to be computable.
- **OWL Full** - This is the most complex sublanguage and supports maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

In terms of terminology, OWL has some concepts that are worth pointing out, as they execute an important role in the work here presented:

- **Individual** - Corresponds to an instance of any given class. Or, in an object-oriented view, it corresponds to and object.
- **Class** - It is a concept that corresponds to a collection of instances (or individuals). In OWL there is a special class, *Thing*, which represents anything. All classes defined in this language are subclasses of Thing.
- **Property** - A directed binary relation that describes class characteristics. Individuals of a given class will use these characteristics as its attributes. Properties may possess logical capabilities such as being transitive, symmetric, inverse and functional.
- **Datatype Property** - A subset of properties that relate individuals of a given class to RDF literals or XML schema datatypes, like integers, booleans or Strings.
- **Object Property** - Since datatype properties only relate individuals to datatype values, there must also exist a subset of properties that handles relations between individuals, which is exactly what object properties do.

A simple example of an OWL ontology can be seen in Listing 3.1. This example states that the individuals *bill* and *alice* are both of the class *Person* and that the individual *alice* has an object property *hasSon*, which points to the individual *bill*. Both have a *name* property which, in the case of *alice* has the value *Alice* and in the case of *bill* has the value *Bill*. To simplify, this ontology means that Bill and Alice are both persons and that Bill is Alice's son.

Listing 3.1: An example of OWL, showing an ontology meaning that Bill and Alice are persons and that Bill is Alice's son.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://www.fct.unl.pt/ontologies/example.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://www.fct.unl.pt/ontologies/example.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Person"/>
  <owl:ObjectProperty rdf:ID="hasSon">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Person"/>
  </owl:ObjectProperty>
  <owl:FunctionalProperty rdf:ID="name">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Person"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  </owl:FunctionalProperty>
  <Person rdf:ID="alice">
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Alice</name>
    <hasSon>
      <Person rdf:ID="bill">
        <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >Bill</name>
      </Person>
    </hasSon>
  </Person>
</rdf:RDF>
```

OWL is a full-fledged ontological language with hundreds of key-words and possibilities. It provides a great deal of expressiveness. Further examples, references or details can be found in the OWL reference site [W3C, 2004b].

3.2.3 SPARQL

Much like Structured Query Language (SQL) for databases, a query language for RDF exists by the name of SPARQL Protocol and RDF Query Language (SPARQL). It can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The results of SPARQL queries can be results sets or RDF graphs.

Since OWL is a subset of the RDF language, SPARQL is also compatible with ontology graphs, which is very useful in the context of this thesis, although it is not optimized to query OWL. To overcome some of the limitations SPARQL-DL was created [Sirin and Parsia, 2007], which natively queries OWL-DL.

SPARQL has a very expressive syntax that allows the queries to, for example, limit their results or sort them in ascending or descending way, along with many other possibilities, much like in SQL. Beyond that, it has four query forms to retrieve information from a RDF graph: *SELECT*, *CONSTRUCT*, *ASK* and *DESCRIBE*. In the next paragraphs, these forms will be explained.

The SELECT query returns all, or a subset of, the variables bound in a query pattern match. An example of this query can be seen in Figure 3.2, where a query is being made to the example ontology in Listing 3.1 to select all the persons that have a son along with the respective sons. This query will yield a result of "Alice" for the variable *person* and "Bill" for the variable *son*.

Listing 3.2: An example of a SELECT query, which returns a person and her son.

```
PREFIX ex: <http://www.fct.unl.pt/ontologies/example.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?person ?son
WHERE
{
        ?person ex:hasSon ?son .
        ?person rdf:type ex:Person .
}
```

The CONSTRUCT query returns a RDF graph constructed by substituting variables in a set of triple templates. The example in Listing 3.3 shows the construction of a new graph using the triples found in the WHERE clause.

Listing 3.3: An example of a CONSTRUCT query, which returns a graph containing all the sons and respective parents.

```
PREFIX ex: <http://www.fct.unl.pt/ontologies/example.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT
{
        ?son ex:hasParent ?person .
}
WHERE
{
        ?person ex:hasSon ?son .
        ?person rdf:type ex:Person .
}
```

The ASK query returns a boolean indicating whether a query pattern matches or not. As shown in Listing 3.4, the example query will return true if a person that has a son exists in the queried graph.

Listing 3.4: An example of an ASK query, confirming if a person with a son exists.

```
PREFIX ex:    <http://www.fct.unl.pt/ontologies/example.owl#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ASK
WHERE
{
    ?person ex:hasSon ?son .
    ?person rdf:type ex:Person .
}
```

The DESCRIBE query returns a RDF graph that describes the resources found. In Listing 3.5, the query will return a graph containing the description of all the persons in the queried graph.

Listing 3.5: An example of a DESCRIBE query, which returns a graph containing all the triples of a given person.

```
PREFIX ex:    <http://www.fct.unl.pt/ontologies/example.owl#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

DESCRIBE ?person
WHERE
{
    ?person rdf:type ex:Person .
}
```

SPARQL is a very dynamic query language that offers many possibilities to get information from RDF graphs. Further examples, references and details can be found in the SPARQL submission page [W3C, 2008a].

3.2.3.1 SPARQL/Update

Another useful feature of SQL is the ability to update database tables. Analogously, SPARQL has a derived language that has that same purpose for RDF graphs. This language is SPARQL/Update. Update operations are performed on a collection of graphs in a Graph Store. Operations are provided to change existing RDF graphs as well as create and remove graphs within a graph store, which is a group of graphs where updates may be made.

Among many query types this language uses, the most common ones are *INSERT* and *DELETE*, which are used to insert and remove data from a graph, respectively. It is also allowed to use update operations aided by standard SPARQL queries and syntax. In Figure 3.6 an example query is shown where Brian is inserted as Alice's husband and Bill is removed as Alice's son.

Listing 3.6: An example of a INSERT/DELETE query, which asserts that Brian is the husband of Alice. It also deletes the triples stating that Bill is Alice's son.

```
PREFIX ex:    <http://www.fct.unl.pt/ontologies/example.owl#>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
INSERT DATA
{
        ex:Brian ex:husbandOf ex:Alice .
}
DELETE DATA
{
        ex:Bill ex:hasParent ex:Alice .
        ex:Alice ex:hasSon ex:Bill .
}
```

SPARQL/Update is as expressive as SPARQL and also has many other features. Its implementations are limited, since it is used in few frameworks other than Jena, which was used in this thesis and will be described in the next section. Further examples, references and details can be found in the SPARQL/Update submission [W3C, 2008b].

3.2.4 Jena

In the scope of this thesis, a programmatic bridge between Java and both OWL and SPARQL was necessary. The Jena framework makes this bridge, by supporting RDF, RDFS, OWL and SPARQL, and allowing the integration of these languages with any Java-based framework, such as JADE.

This framework holds information of both RDF and OWL graphs in containers called *models*. In the specific case of OWL, the models are more complex, containing more detailed access methods to manage ontologies without, however, changing the representation originally created in RDF triples.

One of the biggest reasons for creating ontology-based applications is to apply the use of a reasoner in order to derive more knowledge from the information a model already contains, adding it to the knowledge the model already contains. An example of reasoning could be, for the ontology shown in Figure 3.4, if *alice* is declared of the type *Human*, then a reasoner might also conclude that *alice* is of the type *Organism*.

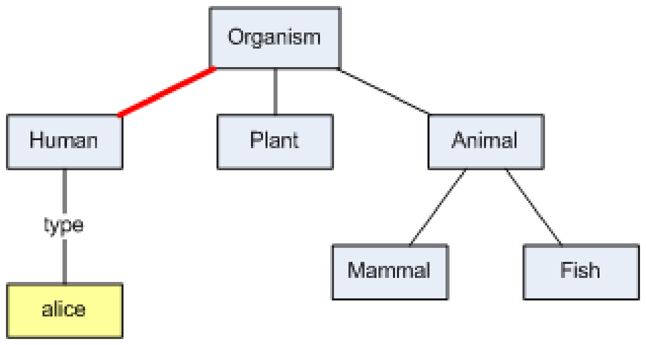


Figure 3.4: An example of reasoning, where the new knowledge is represented by the red connection, stating that if the individual *alice* is a Human, it also is an Organism.

SPARQL is also very important in the context of this work and Jena supports it, along with SPARQL/Update. Queries are made directly to the RDF graph contained in a model and the returned information comes in a form depending on the query type: for a *SELECT*, the information comes in the form of a result set, for a *DESCRIBE* or a *CONSTRUCT*, a model is returned and for an *ASK*, the query merely outputs a boolean value. Any of these information types are compatible within Jena, allowing exchange of information between models.

Thus, the model system in the Jena framework can be represented as shown in Figure 3.5; The basis is the RDF graph, where the information is contained and queries are made. Above, the reasoner, which adds new knowledge to the graph. Finally, on top, the ontology model containing an interface with ontology-focused methods, simplifying queries and knowledge update.

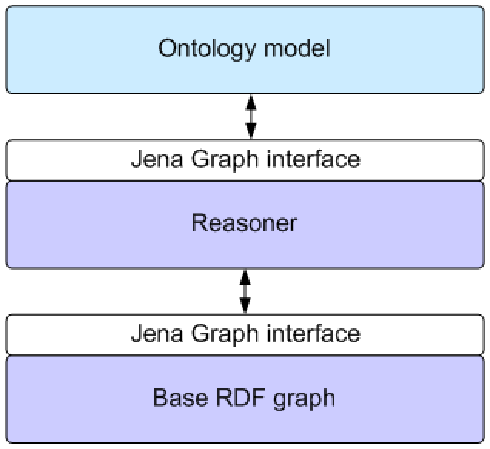


Figure 3.5: The Jena ontology model structure, from [Hewlett-Packard, 2010]

Jena has much more concepts and utilities implemented, such as conversion from RDF to Java or permanent storage for ontologies, so for more information about using this framework, its home website should be consulted [Hewlett-Packard, 2010].

3.2.5 Protégé

Protégé is a free program for editing ontologies, with support for RDF and OWL, developed by the Department of Medical Informatics, Stanford University [Gennari et al., 2003]. Even though it has been developed for biomedical applications, the system may be used in any other areas.

The architecture of Protégé is separated into two components. The component module (Protégé Application Programming Interface (API)) and the component view (Protégé GUI). The first component is located in the mechanism of internal representation of ontologies and KBs. The objective of the second component is to interface with the user in a visual fashion for easy editing and manipulation.

Even though the API does basically the same thing as the, previously described, Jena API, the choice was not to use this one since it is not directly based on RDF and

OWL from the start as Jena is, which makes implementations more difficult. On the other hand, the Protégé GUI was very useful for visually constructing ontologies, since it contains class, property and individual editors, along with the support for plugins that aid in the visualization of the ontological descriptions. This graphical environment can be seen in Figure 3.6.

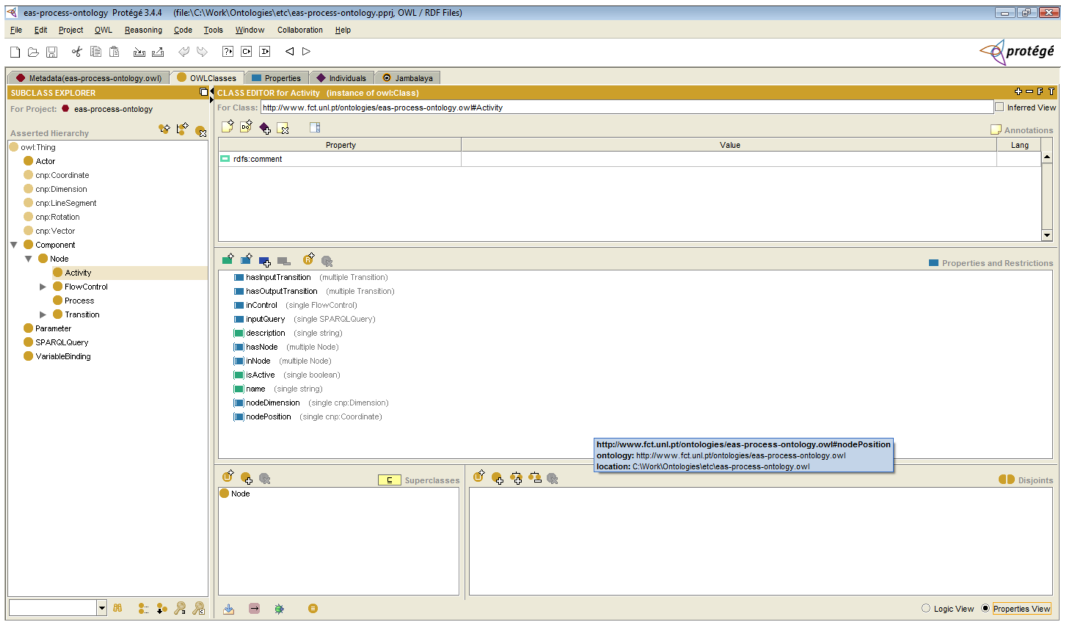

eas-process-ontology Protégé 3.4.4 (file:\C:\Work\Ontologies\etc\eas-process-ontology.pprj, OWL / RDF Files)
File Edit Project OWL Reasoning Code Tools Window Collaboration Help
protégé
Metadata(eas-process-ontology.owl) OWLClasses Properties Individuals Jambalaya
SUBCLASS EXPLORER
For Project: eas-process-ontology
Asserted Hierarchy
owl:Thing
Actor
cnp:Coordinate
cnp:Dimension
cnp:LineSegment
cnp:Rotation
cnp:Vector
Component
Node
Activity
FlowControl
Process
Transition
Parameter
SPARQLQuery
VariableBinding
CLASS EDITOR for Activity (instance of owl:Class)
For Class: http://www.fct.unl.pt/ontologies/eas-process-ontology.owl#Activity
Inferred View
Annotations
Property
Value
Lang
rdfs:comment
Properties and Restrictions
hasInputTransition (multiple Transition)
hasOutputTransition (multiple Transition)
inControl (single FlowControl)
inputQuery (single SPARQLQuery)
description (single string)
hasNode (multiple Node)
inNode (multiple Node)
isActive (single boolean)
name (single string)
nodeDimension (single cnp:Dimension)
nodePosition (single cnp:Coordinate)
http://www.fct.unl.pt/ontologies/eas-process-ontology.owl#nodePosition
ontology: http://www.fct.unl.pt/ontologies/eas-process-ontology.owl
location: C:\Work\Ontologies\etc\eas-process-ontology.owl
Superclasses
Node
Disjoints
Logic View
Properties View

Figure 3.6: The Protégé GUI.

3.3 Business Process Management

As stated before, enterprises need to adapt quickly to new demands in order to keep competing in today's market. To deal with these types of challenges, companies have started to use Information Technology (IT) to manage business processes. As the time went by, business planning replaced paper by computers, in what was later known as Business Process Management (BPM) [Ryan K.L. Ko, 2009].

A definition is needed for BPM an many literature items are in agreement that it consists in [Ryan K.L. Ko, 2009]:

> *Supporting business processes using methods, techniques and software to design, enact, control and analyse operational processes involving humans, organizations, applications, documents and other sources of information.*

The BPM concept is used in what is known as a Business Process Management System (BPMS), which can be defined as [van der Aalst et al., 2003]:

A generic software system that is driven by explicit process designs to enact and manage operational business processes.

The basic principle of BPM, as the name states, is the use of processes and their application in the business world. Processes are very general concepts. Keeping that in mind, throughout this work, the considered definition of a process is the one cited below [Davenport, 1993]:

... a process is simply a structured, measured set of activities designed to produce a specified output for a particular customer or market. It implies a strong emphasis on how work is done within an organization, in contrast to a product focus's emphasis on what.

BPM enables businesses to respond to changing consumer, market, and regulatory demands faster, creating competitive advantage. To be able to have such a quick adaptation to changes, it is always in constant change within a certain company. This is due to the fact that BPM activities form a life-cycle. This brings with it the benefit of being able to simulate changes to business processes based on real-life data. Also, the coupling of BPM to industry methodologies allows users to continually optimize the process to ensure that it is tuned to its market needs. The BPM life-cycle describes the various phases in support of operational business processes. The phases are shown in Figure 3.7 [van der Aalst et al., 2003].

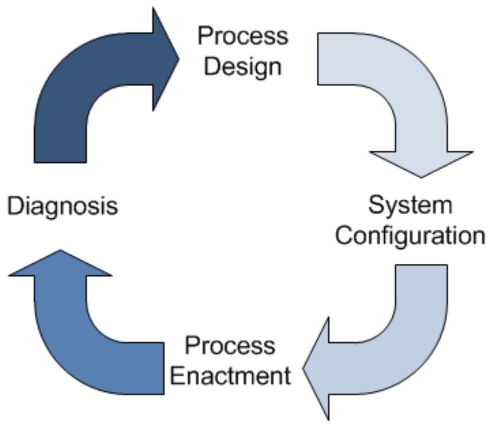


Figure 3.7: The BPM life-cycle. It consists of four stages: *process design*, *system configuration*, *process enactment* and *diagnosis.*

As shown in the figure, the BPM life-cycle consists in four stages: *process design*, *system configuration*, *process enactment* and *diagnosis*. These are described below.

- **Process Design** - In this stage, the various processes are identified and a flow between them is designed by using tools for the effect.
- **System Configuration** - The BPMSs are configured, roles are analysed and variables are introduced to the processes.
- **Process Enactment** - The configured BPMSs are deployed into engines that will execute them whenever requested to do so.
- **Diagnosis** - The BPMSs are analysed and statistics are taken, using tools for that purpose, in order to be improved when the cycle reaches the process design stage again.

BPM is, therefore, very useful to define enterprise functioning stages as processes. In the case of manufacturing, processes, like product development, are made of activities that require certain skills to be accomplished. Product designs are generated by research and development, tested for market and evaluated by manufacturing or engineering as shown in Figure 3.8 [Davenport, 1993].

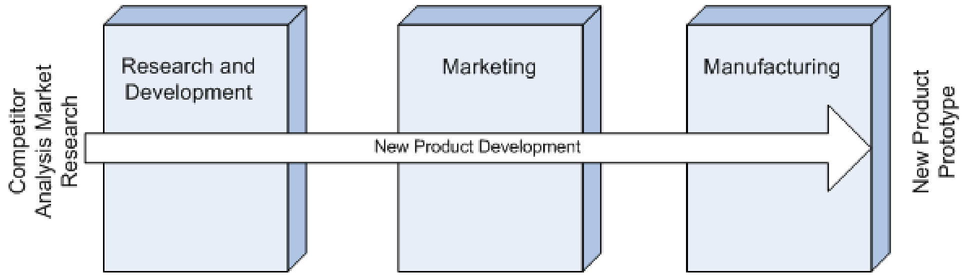


Figure 3.8: A cross functional process. Adapted from [Davenport, 1993]. A process of a new product development requires research and development, marketing and manufacturing to achieve a new prototype.

If an organization is viewed in terms of processes from the top to the bottom of the chain, cost reduction and quality improvement can be some of the advantages.

BPM derives from Business Process Reengineering (BPR), which is a concept that arose in the 1990s whose objective was to help companies to cope with the constantly changing environment. BPR strived for the perfect process definition model a company could use, re-organizing the whole functioning whenever it was not viable. This radical view was not very practical. BPM has taken the advantage of the BPR experience and conceptually is more flexible in terms of extensibility and intensity. Unlike BPR, which targets end-to-end process by radically redesigning it, BPM can be applied part by part to the whole enterprise at a time, by adopting much more practical, iterative and incremental changes in business processes [Ryan K.L. Ko, 2009].

3.3.1 BPM vs. Workflow Management

Business Process Management (BPM) and Workflow Management (WfM) are two concepts often used together [Ryan K.L. Ko, 2009]. There are two main viewpoints regarding the differences between these two subjects. The first is that BPM should be considered a discipline, with WfM providing the technological support for its implementation [Hill et al., 2008]. The second, and accepted by most studies, is that features in WfM are a subset of the ones in BPM, the main difference being the support for diagnosis in BPM [Georgakopoulos et al., 1995]. However, according to Ryan Ko [Ryan K.L. Ko, 2009], BPMS and Workflow Management Systems are in fact the same, as BPMSs still do not have a mature support for diagnosis.

Regardless of the similarity between BPM and WfM, the author of this thesis decided to base the work in BPM, since the theory behind is more easily applied to its context.

3.4 Conclusions

This chapter provided a basis for the rest of this document, as it introduced several concepts and technologies within the scope of this work. MAS, along with its Java implementation, JADE, was used to create a basic architecture, in which each agent was assigned with specific tasks and responsibilities and, by means of the communication standards referred in this chapter, organized itself with the rest of the system. Ontologies and Jena were implemented on top of the created MAS to provide a language both for communication and for knowledge storage and analysis. Finally, the concept of BPM was used to implement a process engine inside each agent, granting it the power of complex behaviour coupled with easily configured features. This process engine will be detailed further in Chapter 5 and in Section 6.3.

The choice of technologies took in account the possibility of integration between them, along with community usage and support. Also, JADE implemented a well-defined system for agent communication as well as a yellow page service to find agents in the network. Jena, on the other hand, provided support for ontological definitions in OWL and SPARQL, which, in turn, were chosen due to their growing expressiveness and rich syntax, as well as due to the fact that both enabled integration with other technologies, if needed.

In the next chapters, the actual usage of the described concepts and technologies in this thesis will be explained.

Chapter 4

A BPM-Based Architecture for EPS

As stated in Chapter 1, the objective of this thesis was to build a system that coped with changes in the production orders and organization of the machines in terms of layout and cooperation objectives, so that the orders might be executed with optimal or near optimal efficiency. So, the system had to be sufficiently agile in order to deal with the re-organization issues. To achieve this goal, the points of modularity, programming effort reduction, re-usability and self-organization were considered during the planning stages.

This system uses the benefits of reconfiguration and decentralization to reach the above objectives. The reconfiguration capability comes from the idea of BPM where all the tasks in a given company are described by processes, which are, by nature, easily reconfigurable. Also, the use of ontologies for description helps with this feature, along with the need for the common understanding between the system components. Decentralization comes from the fact that the system here presented is a MAS, which distributes the workload by individual agents.

This chapter explains the approach used for the design and description of the overall process-based architecture identifying the several agent types involved in the developed MAS as well as their responsibilities and interactions. It then moves on to the discussion of the EPS sub-architecture, which consists on a configuration of the process-based architecture with processes and ontologies designed to control and describe the components of a manufacturing system.

4.1 Generic Process-Based MAS Architecture

Agility was a major concern while implementing this system. As previously stated, the BPM concept is closely related with agility within enterprises and, therefore, ideal to integrate in this work.

Tools that offer BPM-related solutions are, by nature, easily reconfigurable, which allows companies to adapt quickly to internal or external changes by redesigning the services and products offered or fine-tuning their internal operations. TIBCO ActiveMatrix BPM [TIBCO, 2011a], Microsoft BizTalk [Microsoft, 2011] and Bonita Open Solution [Bonitasoft, 2011] are some of the tools already used by companies.

BPM has a close relationship with Service-Oriented Architecture (SOA), thus generating applications such as Tibco ActiveMatrix BusinessWorks [TIBCO, 2011b] or specifications such as OWL-S [W3C, 2004a]. Also, and as stated in the previous chapter, BPM has a close relationship with workflows. That is why many implementations of BPM engines are based on the XML Process Definition Language (XPDL) standard for workflow specification [WfMC, 2008].

The architecture presented in this thesis took into account the work of the aforementioned tools and specifications, using concepts based on what has already been done, thus introducing a process-based approach to a MAS, while also supporting descriptions in OWL. Due to its BPM nature, this architecture is generic, making it able to be reused in other contexts beyond manufacturing. As an example, a manufacturing company could be process-oriented from its top administrative level to the lowest shop floor level and, therefore, agile from top to bottom. In order to have a more specific system, the agents need to be properly configured. This base architecture is shown in Figure 4.1.

As illustrated in Figure 4.1, the base architecture contains only two types of agents: The Process Agent and the Monitor Agent. These agents will be explained in detail in the following sections. For now it is only necessary to know that the Process Agent is in charge of executing processes and the Monitor Agent is an interface for the human user, allowing creation, destruction, monitoring or changing of the rest of the agents in the system.

All system agents support OWL, whether it is for communication, configuration or knowledge storage. In the case of the Process Agent, for example, its configuration is based solely on this language, meaning that both processes and other descriptions need to be imported using OWL.

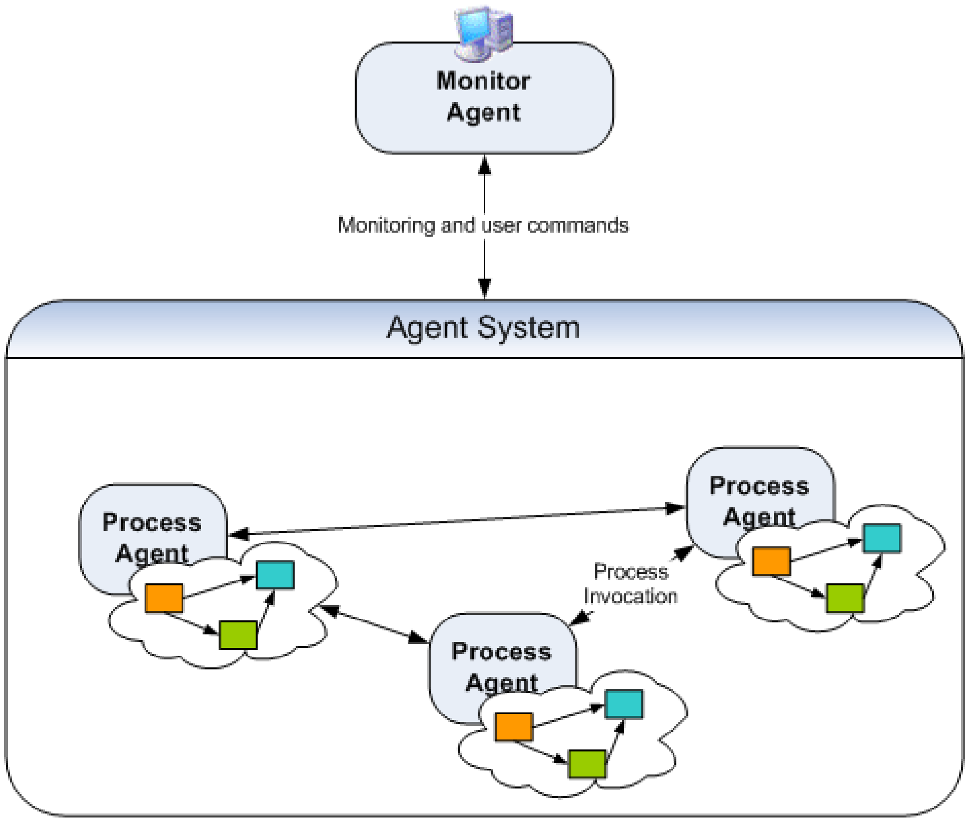


Figure 4.1: The basic architecture of the process-based system, which consists of a Monitor Agent and several Process Agents that execute processes and are able to communicate with each other.

Even though the description of the system is a well-defined set of ontologies created in order to support a great number of concepts (like manufacturing modules), there is always a very high risk of change it in order to support another concept or to create a whole new functionality altogether. In these cases, in many systems, a single change in the ontological description of the classes may require reprogramming or a new system altogether. This is why the system is process-based: any ontological alteration may only require a simple reconfiguration in the executed processes, if any change is needed. Therefore, the agent system is as generic as possible. Each agent can be configured with a certain ontology and, without further programming, can act according the information given.

Ideally, only one type of agent would be needed in such a system, since it may be dynamically configured to do anything. But more specific agents, like the Monitor Agent, may be found useful to the system as they may simplify some roles, while keeping the system generic, as long as they respect concepts that are fixed throughout the system in order to be compatible with it.

In this system, as in any MAS system, it is important for agents to communicate with each other in order to reach a certain goal. It is, therefore, important that agents understand each other. Even though JADE already provides an built-in ontology creation solution for messages, it was decided that all the agents in this system would exchange messages in OWL, SPARQL or SPARQL/Update languages. Since all the agents KBs are constructed through ontological descriptions in OWL format, values described inside may be sent by the messages directly, without any kind of translation to a communication-specific language. Inversely, messages received may be put directly in the KBs in order to be reasoned with or queried.

4.1.1 The Process Agent

The base of the entire system is the Process Agent, which was designed to be as generic as possible. Figure 4.2 shows the inner architecture of this agent.

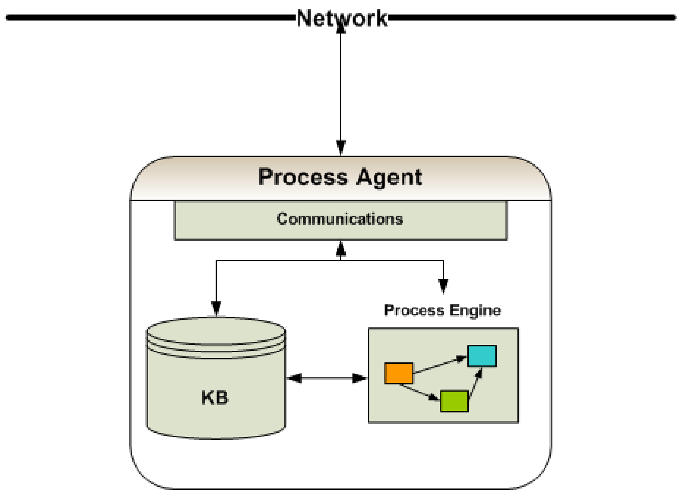
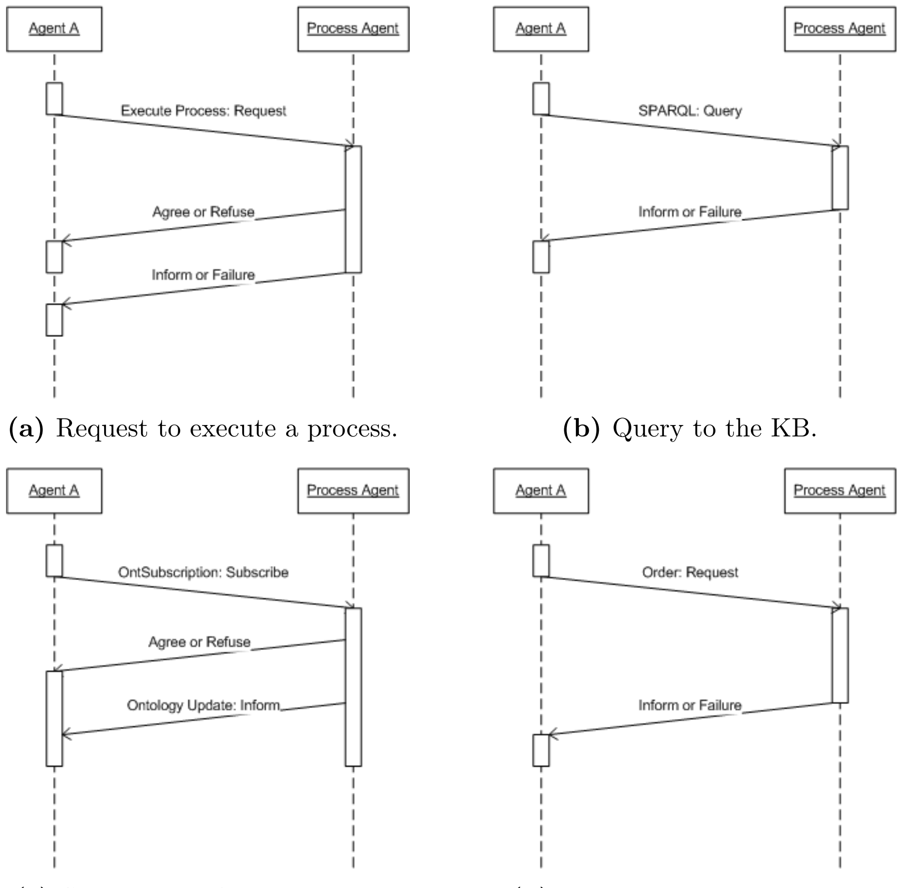


Figure 4.2: The architecture of the Process Agent, which contains a communication layer, a KB and a process engine, all interconnected.

This agent is able to load an OWL ontology to its KB, which may be changed externally if necessary. In order for this agent to support execution of generic behaviours, a process engine was created so processes defined in the ontology can be loaded and executed. This engine enables the agent to have the functionality the system designer aims for without any need for reprogramming. It still has a communication layer, which enables messaging with other agents using OWL, SPARQL or a combination of both. This allows the agent to send orders or queries to other agents keeping its surrounding environment perception up-to-date.

In the architecture, the communications component is directly linked to the KB component, allowing direct KB updates or for the agent to be queried for knowledge contained in it. The third component is the process engine whose responsibility is to load and execute processes. This engine is directly connected to the KB so that it can execute the processes based on information already existing in the ontologies. It is also directly connected to the communications component. This allows the loaded processes to evaluate messages from or send messages to other agents.



(a) Request to execute a process. **(b)** Query to the KB.

(c) Subscription for KB update. **(d)** Request to execute orders.

Figure 4.3: The default communication interactions of the Process Agent, based on the FIPA standards.

Because this is a generic agent, it may support any FIPA communication protocol as long as the processes loaded support it. Despite this, a certain number of default interactions were developed so that the agents support operations like executing external processes, updating or querying the KB and subscribing to get all the ontological information of the KB. These interactions can be seen in Figure 4.3.

In the proposed system, processes inside a given Process Agent may be invoked remotely. The exchange of messages depicted in Figure 4.3a exemplifies how this may be accomplished. By using the full FIPA Request Interaction Protocol[1], the invoker sends a request to execute a process, as well as other relevant data like the input parameters if needed. The Process Agent who received the message may agree or refuse to execute, depending on the data received in the request. If the execution is agreed, then the Process Agent sends the resulting information, like the outputs generated by the process. If, for any reason, the execution failed, the Process Agent also informs the invoker of the failure, along with the associated errors.

In many cases there may be a need to change, at run-time, the ontological descriptions of a given agent like, for example, to manually recover from errors. There may also be a need to query the KB in the agent for more information. In order to make this possible, support for the reception of SPARQL and SPARQL/Update was inserted in the Process Agents. In this case the FIPA Query Interaction Protocol was used for the message exchange. Thus, as described in Figure 4.3b, the Process Agent receives a query in SPARQL (for ontological queries) or SPARQL/Update (for ontological updates). The agent, after the query is executed, returns an information message. This message contains the response in the case of a query to the KB. If the order was to update the KB of the agent, then the information message simply states that the update was successful. If, for any reason, the query or update execution failed, the Process Agent returns a failure message.

It may, sometimes, be necessary to make some sort of debug to the KB of Process Agents. For this, an external agent may require access to it in order to analyse the stored information. In this case, the FIPA Subscribe Interaction Protocol was used so that other agents could access the needed information, as can be seen in Figure 4.3c. For this exchange

[1]The reason why a *full* FIPA Request Interaction Protocol is mentioned is because JADE provides simpler versions certain protocols, which take out the agreement or refusal part, resulting in less exchanged messages and providing a faster interaction between agents.

of messages, the Process Agent receives a subscription message. Upon receiving this message, the agent may agree to the subscription, sending an agreement message, along with all the data already stored in the KB, or refuse by sending a refusal message. If the Process Agent has agreed to the subscription, whenever its KB is changed, it sends an information message with the appropriate update, stated in the SPARQL/Update language.

Process Agents also need support for orders so that they may execute some pre-defined behaviours, like deactivation, for instance. Orders may be sent using a simpler version of the FIPA Request Interaction Protocol that is provided by JADE. This version takes out the agreement/refusal message from the protocol to speed up the exchange of messages. Therefore, the Process Agent simply receives a request with the intended order and replies with an information message if it was successfully executed or a failure message otherwise, as shown in Figure 4.3d.

There is a very tight relationship between all the components contained in the architecture of the Process Agent. Information is constantly exchanged between them, as can be seen in Figure 4.4.

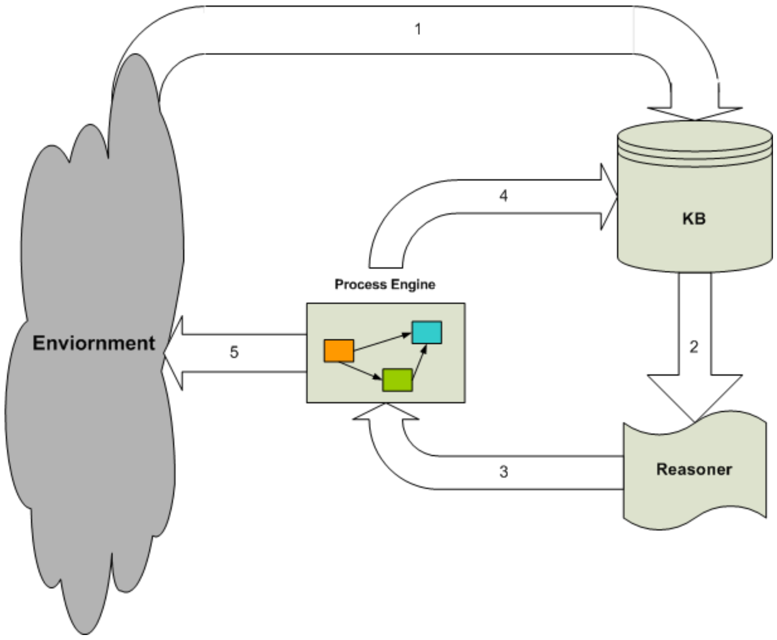


Figure 4.4: Information flow inside the Process Agent. **1**-Information is received from the external environment. **2**-The information is caught by the reasoner, which draws further conclusions. **3**-The process engine uses the information during execution. **4**-The process engine may update the KB with further information. **5**-The process engine may communicate with the external environment.

The previous figure shows the several steps of informational flow within the Process Agent. In step 1, the information perceived from the surrounding environment is received and inserted in the KB. This information is caught by the attached reasoner, as seen in step 2, which draws the adequate conclusions. The process engine uses the information outputted by the reasoner, in step 3, to execute the stored processes. The processes can assert new knowledge to the KB, through the connection in step 4, or execute actions in the surrounding environment of the agent, as shown in step 5.

The KB component is the second-most important part of the architecture and is responsible for storing all the knowledge of the agent and deducing new knowledge, through the reasoner, giving it a certain intelligence. In every case where a KB is inserted in the architecture, it always has an attached reasoner to infer new knowledge based on the one already stored, as described in Chapter 3.2.4. The KB is dynamic, since the information about the agent and the surrounding environment is in constant change. Also, the information may be changed due to new concepts arriving or the system designer concluding that a new description is better than the previous one. This implies that the KB and agent programming must be dynamic enough to cope with the changes. In order fulfil this demand, the agent is capable of updating the KB on-the-fly and of generating new conclusions.

The KB component and the process engine component have a very tight relationship, since the processes are described in OWL and loaded directly from the KB. Also, the engine must constantly query the KB to get all sorts of information. This process engine is the most important component in the architecture, being the core of the Process Agent and controlling its actions by means of the loaded processes. It allows the agent to assume any behaviour loaded in the KB using the ontologies. This makes each Process Agent fully configurable and removes the need to hard-code the agent behaviours. This component uses the BPM paradigm to execute the behaviours. Each process contains a set of activities that are executed in a given order. The activities may execute a certain action inside the agent, influence the outside environment or assert or query facts from the KB. Since this is the most important component in the whole architecture, Chapter 5 was created specifically to explain the process model behind it.

To keep ontological descriptions coherent without any contradictions throughout the system, each Process Agent is solely responsible for its own KB information. Event though a use of a central agent to store all the ontologies in the system could be useful, this might generate incoherences due to the fact that information in the Processes Agents is in constant evolution. Also, if the central agent were to be shut down for any reason, the system would be extremely crippled. For this purpose if it is necessary to query the information of a given agent, this query is made directly to it instead of using a central repository, as shown in Figure 4.5.

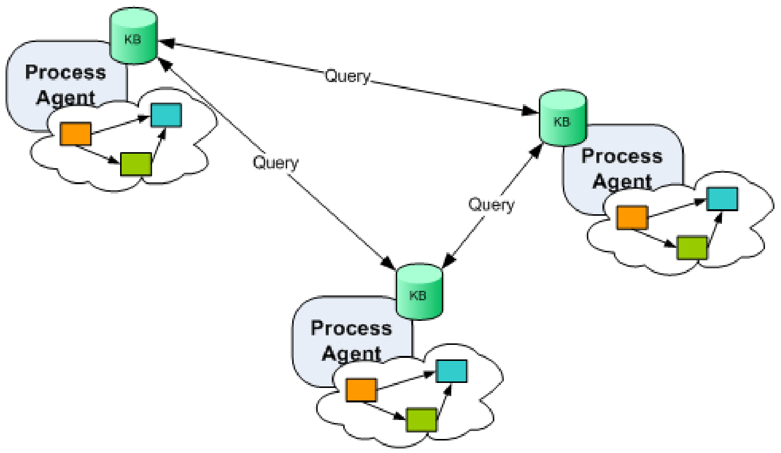


Figure 4.5: External queries between the Process Agents. This allows the system to be completely distributed, without the need for a central entity to manage the knowledge.

This scattering of information is made possible by the fact that all the agents possess a phantom KB that represents the common ontology of the whole system. This KB itself does not contain any information, instead, when any query inside the agent is directed to this KB, it will result in an exchange of messages with the appropriate agent containing such information, using the FIPA Query Interaction Protocol, in Figure 4.3b. Agents are also able to insert knowledge in the KB of other agents, by updating this phantom KB. In order to know where to send the queries or updates, each agent possesses a list of all the other agents in the system. All the agents are subscribed to the JADE DF agent, which updates their lists whenever a new agent enters the system.

4.1.2 The Monitor Agent

Much like in the definition of BPM, this architecture supports process diagnosis, by using a third agent dedicated to it. This agent is useful for the system designer to monitor the current state of the system, by showing and processing all the information contained in the Process Agents. This agent is able to keep track of ontologies of the other agents by storing their updates in its KB and reasoning over them. It is also capable of directly changing the ontologies of other agents so that the designer may correct the values he chooses. Its inner architecture can be seen in Figure 4.6.

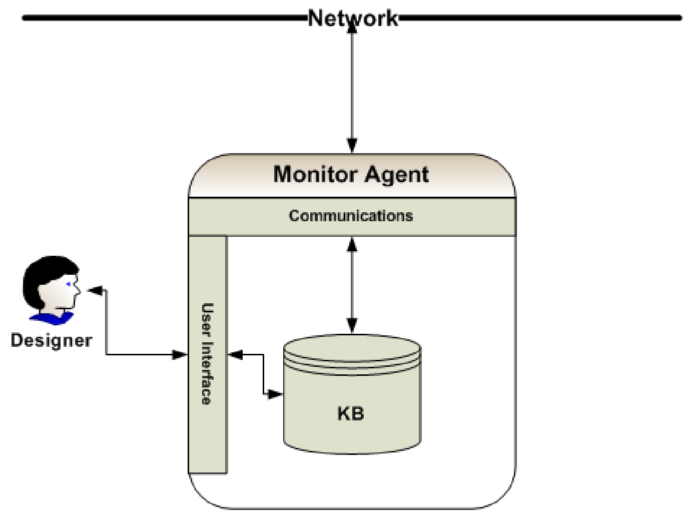


Figure 4.6: The architecture of the Monitor Agent, containing a communication layer, a KB to store the external information and a user interface.

In terms of communications, this agent supports all the message exchanges defined for the Process Agents and shown in Figure 4.3. These are supported so that the Monitor Agent is able to be informed of the contents of all the KBs in the system, alter information in the Process Agents, execute processes for debug purposes and send orders to, for example, shut down agents.

The Monitor Agent also has a KB with an attached reasoner to store the information of the agents it monitors. All information from the user interface is taken from this KB. The visual interface also allows a designer to perform system actions such as creating or destroying agents, update their KBs or view the status of running processes.

4.2 EPS System Architecture

The previously described architecture is flexible enough to control a MAS in a wide variety of domains, since the agents can be configured with any set of processes a designer deems fit to his needs. In the case of this thesis, the system was configured to control a manufacturing shop floor using the EPS paradigm.

The core concepts adopted by many EPSs already developed are *Modules* and *Skills*, which are described in many works about the subject [Onori et al., 2005, Frei et al., 2010, Maraldo et al., 2006, Barata and Onori, 2006]. In this work, manufacturing equipment (like drillers, conveyors, grippers or cranes) and aggregates of these (a cell able to perform a transport of a product to and from a drilling location) can be seen as modules, while the ability of each module to perform tasks, like drilling or transporting, can be seen as skills. This approach is similar to the one presented in previous chapters about EPS and can be seen in Figure 4.7.

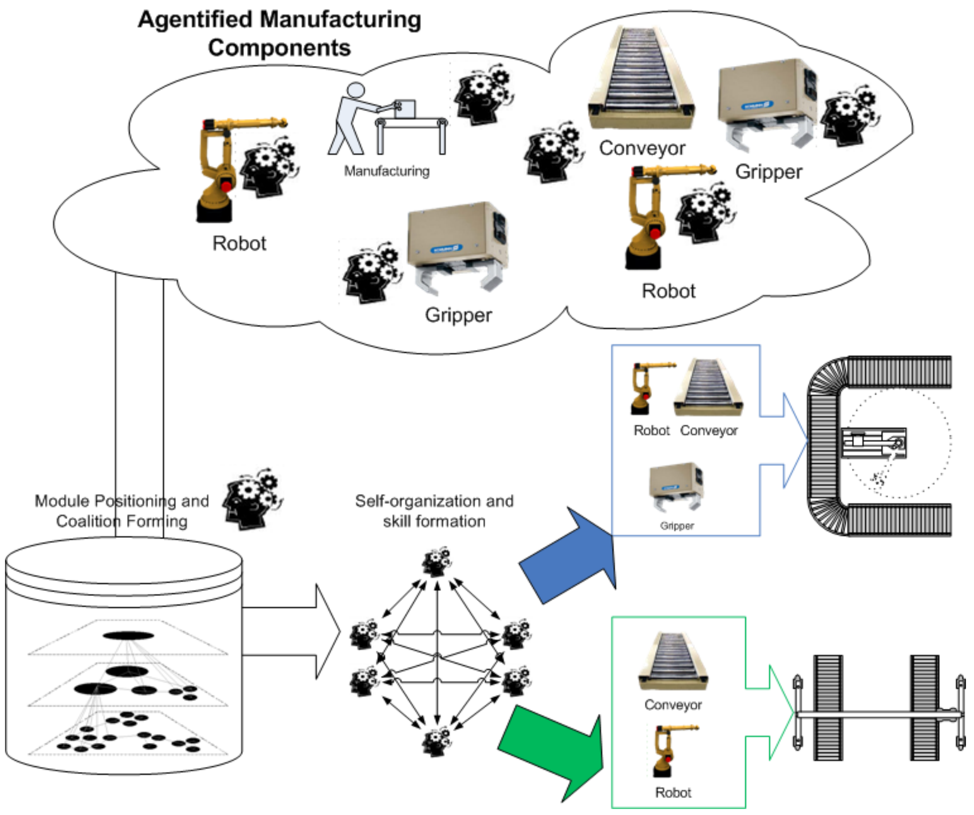

Agentified Manufacturing Components
Manufacturing
Conveyor
Gripper
Robot
Gripper
Robot
Robot
Conveyor
Gripper
Module Positioning and Coalition Forming
Self-organization and skill formation
Conveyor
Robot

Figure 4.7: The base EPS architecture. In this architecture, the manufacturing components are controlled by agents. These components are positioned within the shop floor and organized among themselves to perform manufacturing actions.

Some of the principles in this architecture were based in the CoBasa system, which was already described in Section 2.4. A set of modules represented by agents is placed in the system. Another agent acts as a broker as well as a re-organizer to place the modules in the shop floor. This agent is responsible for dynamically positioning the modules according to their skills and to the manufacturing plans in the shop floor. After the positioning of the agents, they organize themselves in order to be aware of partners with which they will execute skills. This process originates, as a result, several cells that perform certain tasks inside the manufacturing plant.

The agent responsible for positioning the modules in the plant is not in the scope of this thesis. It was developed in another master thesis, using Genetic Algorithms (GAs) approach to achieve near-optimal module positioning [Pereira, 2011]. On the other hand, the objective of this work was to describe and control the agents representing modules in a re-configurable way.

Picking up on the work described in the previous section, Process Agents can be used to represent manufacturing modules, while skills can be a special subset of the process model the agent contains. This is described in Figure 4.8.

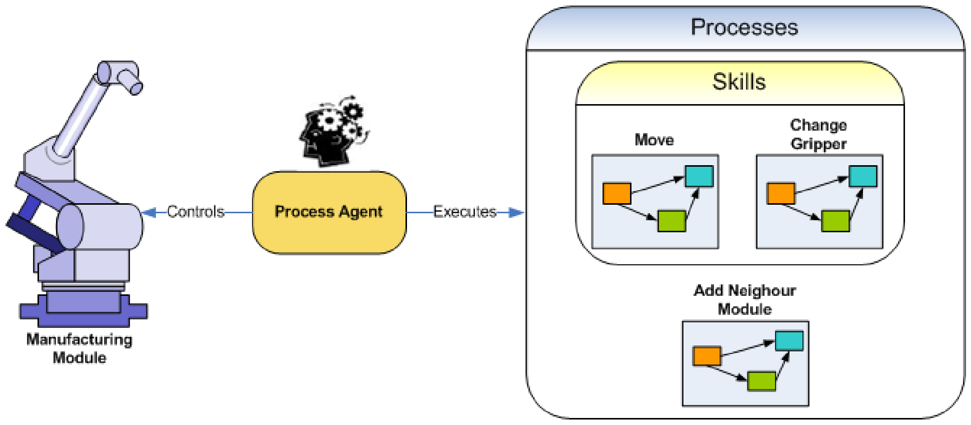


Figure 4.8: A Process Agent configured to control a manufacturing module. This agent contains processes that are used to execute skills of the component.

Since the Process Agents are configurable in OWL, it is necessary a description of the modules in this language. Although a manufacturing ontology was created from scratch for this EPS architecture, this kind of configuration is compatible with the trends of the new agile systems, as many already are described in OWL [Lemaignan et al., 2006, Merdan et al., 2008, Lin et al., 2011].

In the previous section, it was stated that Process Agents were able to request execution of external processes, via the FIPA Request Interaction Protocol. This provides a very useful feature in this particular architecture, allowing modules to use their own skills in conjunction with skills of other modules. An example of this is shown in Figure 4.9.

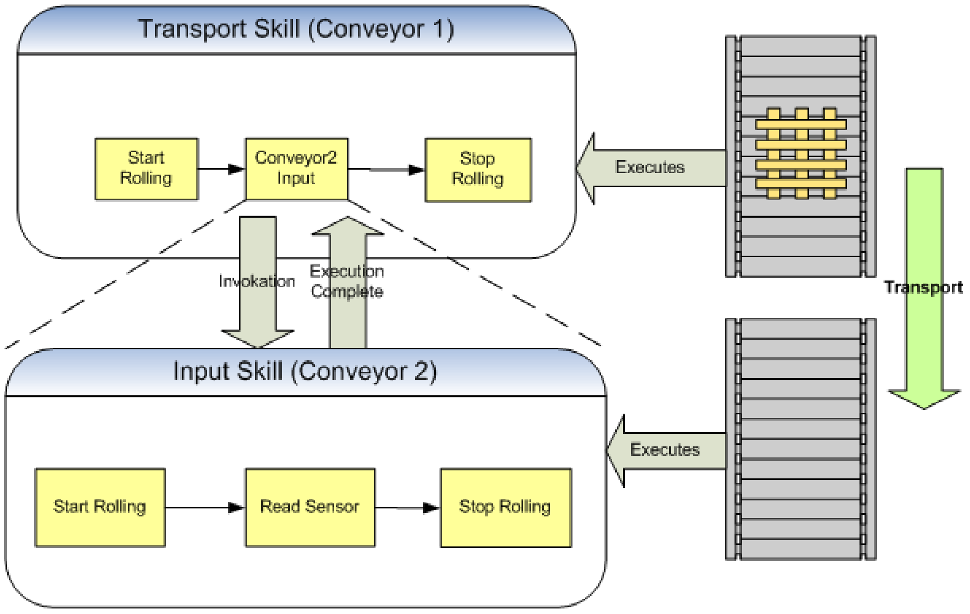


Figure 4.9: An example of external skill execution, where two conveyors must transport a pallet between them. The first conveyor requests the execution of the reception skill of the second conveyor.

In the case of Figure 4.9, to transport a pallet between two conveyors, the conveyor containing the pallet starts moving its belt and orders the receiving conveyor to execute the correct skill in order to receive an object. According to many works, a skill like this can be considered a *Complex Skill*, since it involves a coordination between two or more agents [Maraldo et al., 2006, Barata and Onori, 2006]. Complex skills are described as skills *dynamically formed* from several simpler skills. Even though, in the case of Figure 4.9, two skills are being executed to perform higher-level action, this is not considered a complex skill for the following reasons:

- This skill is still simple enough to be managed by the two agents containing the simpler skills without the need of a *Coalition Leader*, which is used in many other works [Barata et al., 2005, Maraldo et al., 2006].

- To form a complex skill, in theory, the process should be able to dynamically pick several skills and create a new sequence of actions, permitting the skill to be executed. The complex skill would not be known a priori. This would require a very complex algorithm to form the new, higher-level skill.

- In these cases the designer already knows most of the skills a certain module can perform with the help of other modules. It is much easier and reliable to just let the designer create these skills, which is what is done in most works [Barata et al., 2005].

Because of the above reasons, in the current architecture there are no complex skills. All skills that require one or more modules are pre-configured to automatically search for the correct module and skill to be performed. In the example shown in Figure 4.9, conveyors *know* a priori they need another conveyor to perform the transport skill with. Upon deployment, they search in their neighbourhood for other conveyors that are able to perform it and insert that information in their KB. Whenever the transportation skill is requested, the conveyor sees whether it has an available neighbour to perform it with and, if that is the case, the skill is executed.

In the current architecture, modules can be divided into two subclasses in order to be better understood:

- **Unit Modules** - These modules represent a physical equipment contained in the system, like robots or conveyors. These components are able to perform only simple tasks. But, when aggregated to other units, the tasks may be more complex.

- **Cell Modules** - These modules do not represent any physical components but are needed for the organisation of the system. Cells represent a group of modules that work together with the purpose of manufacturing a certain product. These contain a manufacturing plan for the unit modules contained in them and are formed during the placement phase of unit modules in the shop floor. The process of forming such modules is described in the Thesis of Nuno Pereira.

Manufacturing processes in cell modules may be considered complex skills, even though they are not explicitly described like that. These processes can be considered statements that say, for example, *I can input a pallet, drill the products, glue the products and output it back*. These types of process are much more complex that skills provided by unit modules.

Units, in the current architecture, were further divided into *Material Handling Units* and *Transforming Units* for better control over the system. Material handling units are solely responsible for handling the transportation and storing of products. Every action in the system that requires product handling must be performed by one of these units, which can be conveyors or buffers, among others. A transforming unit performs some type of operation on the product that may alter its physical properties. An example of a transforming unit is a driller machine.

With cell modules and unit modules introduced, the control of the system will be as in Figure 4.10. This architecture is similar to a modified hierarchical architecture, which was described in Section 2.5.

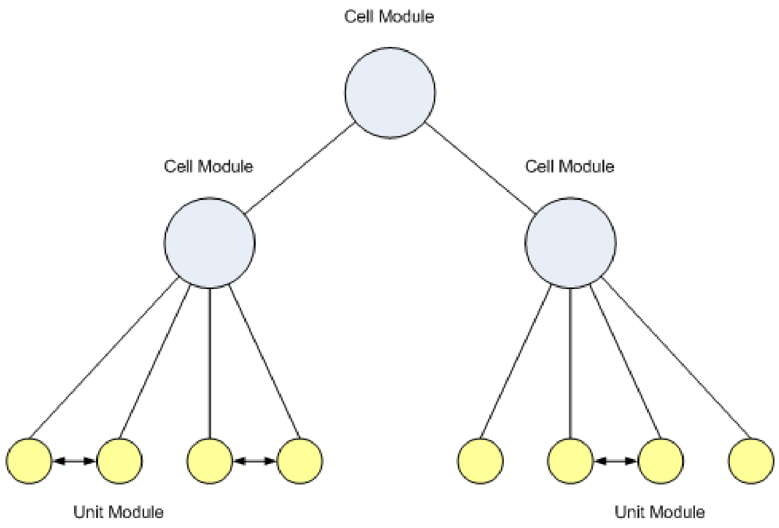


Figure 4.10: The EPS control architecture. Similar to a modified hierarchical architecture.

In the figure, one can see that the cell modules can control more cell modules or unit modules and unit modules can communicate between them to coordinate simple skills that require a few number to execute.

4.3 Conclusions

In this chapter, it was presented the basic architectures for the proposed system. In a lower level, the process-based architecture provides a framework comprised of a MAS able to load and use ontological descriptions, along with a process engine that defines the behaviour of the agents. In the manufacturing context, the EPS architecture presented is merely a configuration of the process-based architecture with ontologies and processes to control manufacturing components.

The conjunction of the two architectures provides an agile system that may easily be modified, since the supporting concept is BPM. The advantage of this approach is that, if the EPS architecture is deemed unfit to cope with certain challenges, it may be reconfigured to work in a completely different way without the need to be hard-coded.

Since the introduction of agility in the manufacturing context, many research has already been made, resulting in several different systems to cope with the problem of adaptation to sudden changes. These systems provided good ideas and solutions. It is the author's belief that the architecture presented in this chapter contributes with new ideas for new systems that may arise.

The heart of both these architectures is the process engine, since it controls the functioning of each agent in the system. This subject will be covered in the next chapter.

Chapter 5

A Process Model Specification

In the context of this work, a specification was required to model the execution and representation of module skills in a generic way to avoid the need for hard-coded skills, which prevent agility. Many skills require more than a simple invocation of their actions, they need a more complex orchestration in order to be dynamic and realistically plausible.

Other constraint was the need to be compatible with the OWL ontology language, the chosen language for the description of the manufacturing modules, and, at the same time, it had to be in conformity with the definition of process and activity already defined in many ontologies for manufacturing systems so that compatibility could be achieved with other descriptions in the same language.

As a result, the following specification was developed in OWL. In order to have a dynamic functionality, the SPARQL language and its subset SPARQL/Update were seamlessly integrated in it. This specification goes along the lines of several other specifications already published in manufacturing-related works [Lemaignan et al., 2006, EUPASS, 2006] as well as existing process and workflow specifications [TIBCO, 2011b, WfMC, 2008, W3C, 2004a], being specifically created for a MAS based on OWL descriptions.

This chapter is totally dedicated to the explanation of the theory behind the process definition used in this work. A process model specification is detailed here and will be further discussed in Chapter 6.3 with the explanation of the engine behind it. This chapter starts by explaining the overall process definition and, afterwards, describes each constituent of the specification. Also, a special section was dedicated to explain the error handling inside processes. Finally, there will be a discussion on other process specifications and technologies.

5.1 Description of the Process Model

In Section 3.3 a process was defined as a structured set of activities. This is what the developed process model aimed for. Structure comes from the use of transitions and flow controls to control the execution of each activity. The structure of a process in the context of this thesis is shown in Figure 5.1.

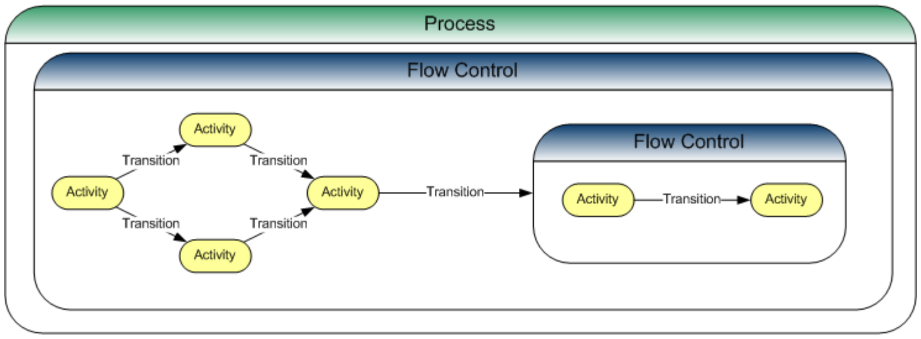


Figure 5.1: The structure of a process, showing that it is composed of activities contained inside flow controls and nodes connected by transitions.

As described in the figure, a process contains activities, which are atomic actions taken in each step of the execution of the process. Activities and other nodes are contained inside flow controls, which control the pace of the execution of the process. Controls may state that a certain set of nodes is to be executed in parallel with another or that the same set may be executed in a loop. Finally the transitions determine the sequence of execution of the nodes contained in the process.

All the components in the figure allow a process to have a complex behaviour, while being able to be easily configured to perform several different tasks. Activities, transitions and flow controls are described in the following sections as well as the process itself.

5.1.1 Process

A process is the main subject in this chapter and the executable component of the engine, which may contain flow controls, transitions and activities inside. A detailed view of a process is described in Figure 5.2.

As shown in the figure, a process has an input, to use certain external values to execute, and an output, that returns a desired set of values that emerged from the execution. To achieve this parameters may be defined for each purpose. Also, if there is need for the

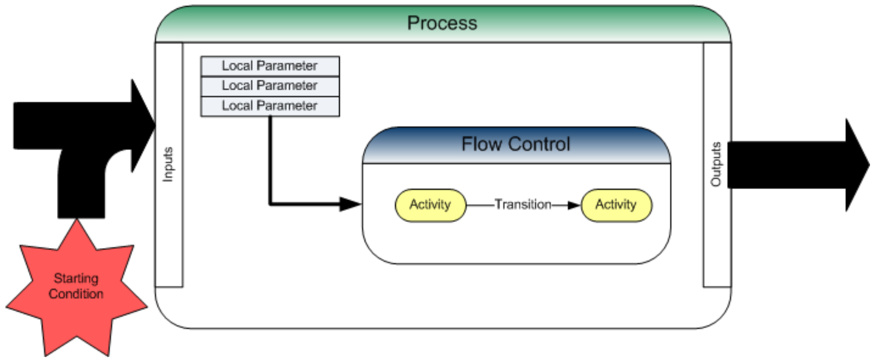


Figure 5.2: A detailed view of a process. It shows that a process may have inputs, generate outputs or contain local parameters. Also, a process may be prompted for execution externally or when a starting condition is evaluated and returns true.

process to hold temporary information during its execution, a set of local parameters can be defined, which are accessible to all the components inside it. When the execution ends, the local parameters are deleted from the memory.

The figure also depicts a starting condition for the process. This condition is a SPARQL query that is evaluated periodically and, whenever it is met, initiates the execution of the process automatically. This allows processes to be treated as rules, in which the *If* clause is the query and the *Then* clause is the process itself.

All processes are associated to certain actors, which means that only the actors containing those processes are able to execute them. Process inside these actors may not be executed by other actors unless a request has explicitly been made to the owner. Figure 5.3 shows the relationship between actors and processes.

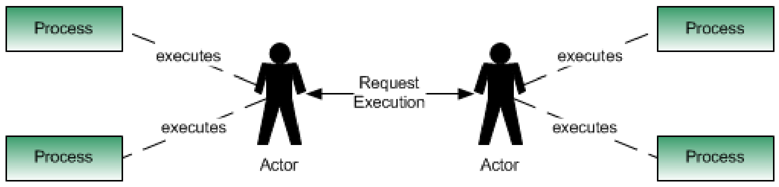


Figure 5.3: The actor-process relationship. Actors are able to execute their own processes or request other actors to execute external processes.

Processes can also be defined as *disabled* in order for the actor not to execute them in certain conditions. This allows for inter-process control, where processes that require a certain process to be disabled may do so. Also, processes may be defined as singleton, which means that there may not be more than one instance running.

As previously stated, the flow controls define the pace of the execution of each node within the process. In the current model, the process itself does not manage activities nor transitions. This task is for the flow controls. This is why, all the processes must define a main flow control, which is the base manager of its execution.

5.1.2 Flow Control

This is a very important type of node in the processes whose sole purpose is to control the execution flow of the activities. They are containers for activities, transitions and even other controls or processes and control how the execution should be scheduled.

In this specification, several types of flow controls where developed to provide possibility for several types behaviours within processes:

- **Sequence** - This control schedules its children to run in sequence. The order of the sequence is determined by the transitions, which will be discussed ahead.
- **Split** - All the children of this class of controls are executed in parallel. No transitions are allowed in a Split.
- **Unordered** - This control executes its children sequentially but in no particular order, therefore transitions are also not allowed in this control.
- **Loop** - This is an abstract control that is used as a template for looping controls. It requires a SPARQL query to be used as an iteration validation. Children of this control are executed sequentially. A number of controls where implemented using this template:
  - **Iterate** - Given an instance that contains a property with multiple values, this control, iterates through all those values, executing once per value and making the current value available for its children.
  - **Repeat Until** - This control repeats its execution *until* a certain condition is evaluated to true.
  - **While** - This control will sequentially repeat its execution *while* a certain condition holds.

Alone, these controls are simple but, since this specification allows controls to be inside controls or connected to other controls, the processes may execute complex flows.

5.1.3 Transition

Transitions dictate the order of sequentially executed flows. To achieve that, transitions must have a source and a target. The source is the currently executed activity, control or process and the target is the next scheduled activity, control or process.

In tree-like processes, where the execution chooses a certain flow depending on a certain condition, conditional transitions need to be used. These types of transitions require a SPARQL ASK query that is evaluated by the engine. If it is evaluated to true, the next node in the transition will be executed. In certain cases, transitions may substitute looping flow controls by creating a loop themselves with the help of their conditions. This may visually simplify the process analysis.

A special type of transition is the error transition, which handles errors from its source node. This transition will be described later in Section 5.2.

5.1.4 Activity

Activities define the specific atomic actions a process may execute. These may need an input, produce a certain output or simply act in a certain way useful for the process. These may range from printing a simple String of text in a console to send a message to another agent. The details of an activity are shown in Figure 5.4.

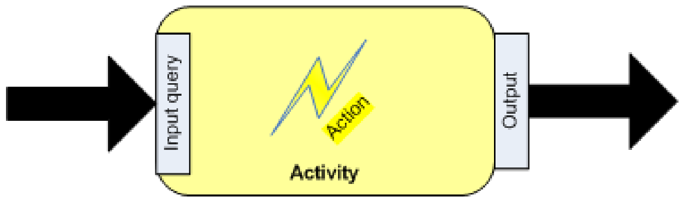


Figure 5.4: A detailed view of an activity. It may require an input or produce an output. In between, a certain action is executed.

In case a certain activity requires a given input for it to be successfully executed, a SPARQL/Update query needs to be defined. This will fill the correct values at runtime. The advantage of this technique is that input parameters may be filled in whichever way the designer wants without any further constraints of information. In the case of outputs, the activities must inject the returning values in the process memory.

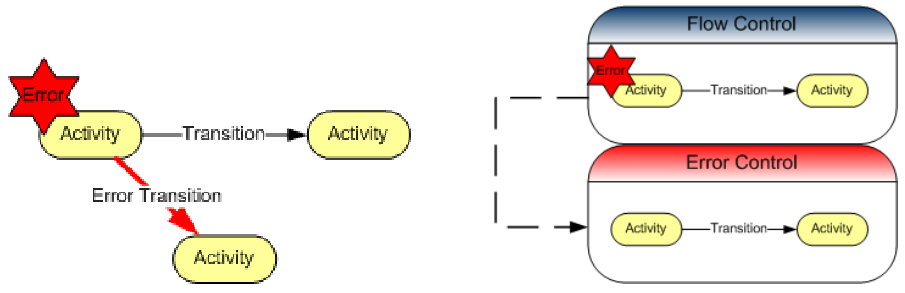
In terms of OWL, an activity is a class that is defined by whoever wants to create a new type of action execution. The designer of the activity does not have to explicitly define which parameters are inputs or outputs. Whenever the activity is executed, the input SPARQL/Update query has to fill the right parameters for execution. This query can also be used to get the correct output parameters from previous activities. Activities and their correct configuration are further described in Section 6.3 and Appendix C.

Since activities are to be implemented externally, when creating a new type a designer must keep in mind that the ontological definition must have a reference to the correct implementation class, so that, when the engine sets up, it will be able to correctly execute the activity.

5.2 Error Control

Sometimes, when an activity is executed, it may fail due to errors generated internally. A designer may prefer to direct the flow of the process in a certain way when these errors appear. From this need arose a special feature of this specification, which is error handling.

Errors in processes may be handled in one of two ways: a per-activity way, where a specific activity is analysed for errors, depicted in Figure 5.5a or a general way, where any error in the process may be caught no matter in which activity it occurred, as shown in Figure 5.5b.



(a) Error control using transitions. **(b)** Error control using flow control.

Figure 5.5: Error control types. The first type requires a transition from the activity that generated the error, while the second type activates an error flow control independently of the activity that generated the error.

For the first case, a special type of transition was created. This error transition connects the analysed activity to the node the designer wants to use to repair the error. This transition is chosen whenever the activity generates an error.

For the second case, a special type of flow control was created. This control is activated whenever an activity, that does not have an error transition, generates an error. This flow control executes sequentially the children nodes inside of it, acting exactly like the sequential flow control.

Activities need to explicitly generate errors and append them to their output in order for the next nodes to have access to the type of error, in case the designer wants to analyse the it and take the necessary actions. When an error occurs, that is not handled, will be appended to the output of the process.

5.3 Conclusions

In this chapter the process model specification used throughout this thesis was explained and its components detailed. Since this subject played a significant role in the implementation of the system, by detailing how agents would behave, it was decided to have a chapter full for its own.

This work was developed so that agents could be reconfigured rather than reprogrammed, so one of the conditions was that the specification needed to be generic enough for that to happen. One of the by-products of this constraint is that the system developed in this thesis may be applied not only in the manufacturing context but also in any context that requires a MAS.

The author does not claim that this is the best solution for the problem of agility in the manufacturing environment or MASs. It is worth noting, though, that BPM approaches have been used for years in other departments of business companies with a great level of success, since it allows them to quickly modify or correct erroneous or obsolete processes, which could be a great asset in the shop floor of a manufacturing company.

Even though many process execution specifications and implementations exist, the author decided to create a new one because the existing did not fulfil certain objectives drawn for this thesis at the time of conception.

JADE already has one implementation for WfM called Workflow Agents Development Environment (WADE) [WADE, 2010]. Though this framework provides process (or workflow) execution functionalities for agents, it has no support whatsoever for OWL. Also, workflows have to be hard-coded into the agents, so reconfigurability would be harder to implement in this system.

There is a specification compatible with OWL that may be used to compose processes, called OWL-S [W3C, 2004a]. This specification is very complex and did not have some of the characteristics planned for this work, like the ability to externally program activities for greater extensibility. Also, this specification was too orientated to WebServices, since its main objective was of invoking them in a certain sequence, and support for ACL would have to be manually integrated in it.

XPDL, created by the Workflow Management Coalition (WfMC), is a widely used standard for workflow and process definition [WfMC, 2008]. The problem with this approach was that it is based in XML instead of OWL, which would result in a more complex implementation of a process engine inside the agents.

The author is aware these and other process specifications are more complete and functional than the one presented here. But, for the stated reasons, they were not used. In Chapter 7.2 further enhancements and ideas are discussed for the current implementation.

Chapter 6

Implementation of the System

In order to validate what has been written so far, a full system was implemented using the concepts, technologies and architectures mentioned in the previous chapters. This system is a MAS, where all the agents use ontological descriptions to configure themselves, communicate and store information. The agents also possess a process engine, that dictates their behaviour in accordance with the processes loaded in their ontologies.

Ontological descriptions and process definitions were created to configure the system to control several manufacturing components within a shop floor. This new configuration aimed to be an EPS, where the modules that represented each component would organize themselves to be able to execute skills together in order to follow a production plan.

This chapter describes the implementation of the proposed architecture in Chapter 4, as well as the implementation of the process engine whose process model was already discussed in Chapter 5. It also explains how both were configured in order to act upon a simple manufacturing shop floor using the EPS paradigm.

The basis of the whole system are OWL ontologies, which were used to configure the agents, so the ones created specifically for this work will be detailed first. Next, it will be described the inner functioning of the process engine as well as ways to configure it with new activities. The Process and Monitor agents will be explained in the following sections, as well as the graphical process editor programmed for the purpose of facilitating process creation. Next, there will be an explanation of proposed EPS, along with the configuration of each module that was created to control specific manufacturing components based on the MOFA kit. Lastly, the test scenarios and corresponding results will be discussed.

6.1 System Ontologies

The whole system runs on the premise that it can be configured via ontological descriptions. These descriptions were written in OWL. This language was chosen because it is highly expressive and has a large user base. Its usage is growing at a fast pace in many applications, including the description of manufacturing systems. For the creation of the ontologies, the Protégé tool was used, since it provides an easy-to-use interface and has many tools to edit an analyse ontologies. Both OWL and Protégé were already described in previous chapters.

Different class ontologies were created for different domains. This domain separation, eases the readability and edition of ontologies. All the domains in this system are interconnected and are imported by each other, as shown in Figure 6.1. This is one of the advantages of OWL, because it allows ontologies to import other ontologies.

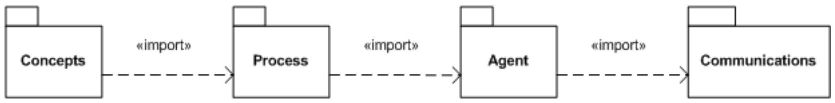


Figure 6.1: The relationship between the different ontology domains in the system. This package diagram shows how the different domains are related.

The considered domains for the core ontological descriptions in the current system were the *Concepts* domain, the *Process* domain, the *Agent* domain and the *Communications* domain. Each of these domains has its own class ontology which will be described along the following sections.

6.1.1 Concepts Ontology

The concepts ontology defines a series of general auxiliary concepts used in all the other ontologies. These are generic concepts that may be found in everyday applications. The URI for this ontology is http://www.fct.unl.pt/ontologies/eas-concepts-ontology.owl. The class diagram for this ontology is shown in Figure 6.2.

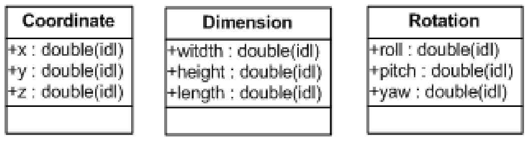


Figure 6.2: The concepts ontology class diagram, showing the three classes that compose it.

6.1.1.1 Coordinate

This class defines a set of Cartesian coordinates to place object or points in a given one, two or three dimensional space. For this purpose, this class has three properties:

- **x** - The X coordinate of a point.
- **y** - The Y coordinate of a point.
- **z** - The Z coordinate of a point.

6.1.1.2 Dimension

The Dimension class defines a given volume of an object in a three dimensional space. It can also define areas, by omitting the length property. This class has three properties:

- **width** - The width of the object or space to describe.
- **length** - The length of the object or space to describe.
- **height** - The height of the object or space to describe.

6.1.1.3 Rotation

The definition of rotations of objects or modules was required in several descriptions in this thesis. This class serves that purpose. It has three properties:

- **roll** - The rotation in the x axis.
- **pitch** - The rotation in the y axis.
- **yaw** - The rotation in the z axis.

6.1.2 Process Ontology

This ontology is the most important in the whole system, as it describes how process descriptions should be created. The whole behaviour of the process engines inside the Process Agents depends on what is defined using this class description. Every process contains instances of these classes which are loaded to the KBs.

Manufacturing ontologies describe manufacturing processes, so, this ontology was planned to be as generic as possible to be able to describe such processes and also expressive enough to define sequences of activities that might actually be executed inside an agent equipped with an engine.

This ontology imports the previous description in the concepts domain. Its URI is http://www.fct.unl.pt/ontologies/eas-process-ontology.owl. Since the associations between classes defined for processes are a bit complex, a simplified version of the Unified Modelling Language (UML) class diagram is shown in Figure 6.3.

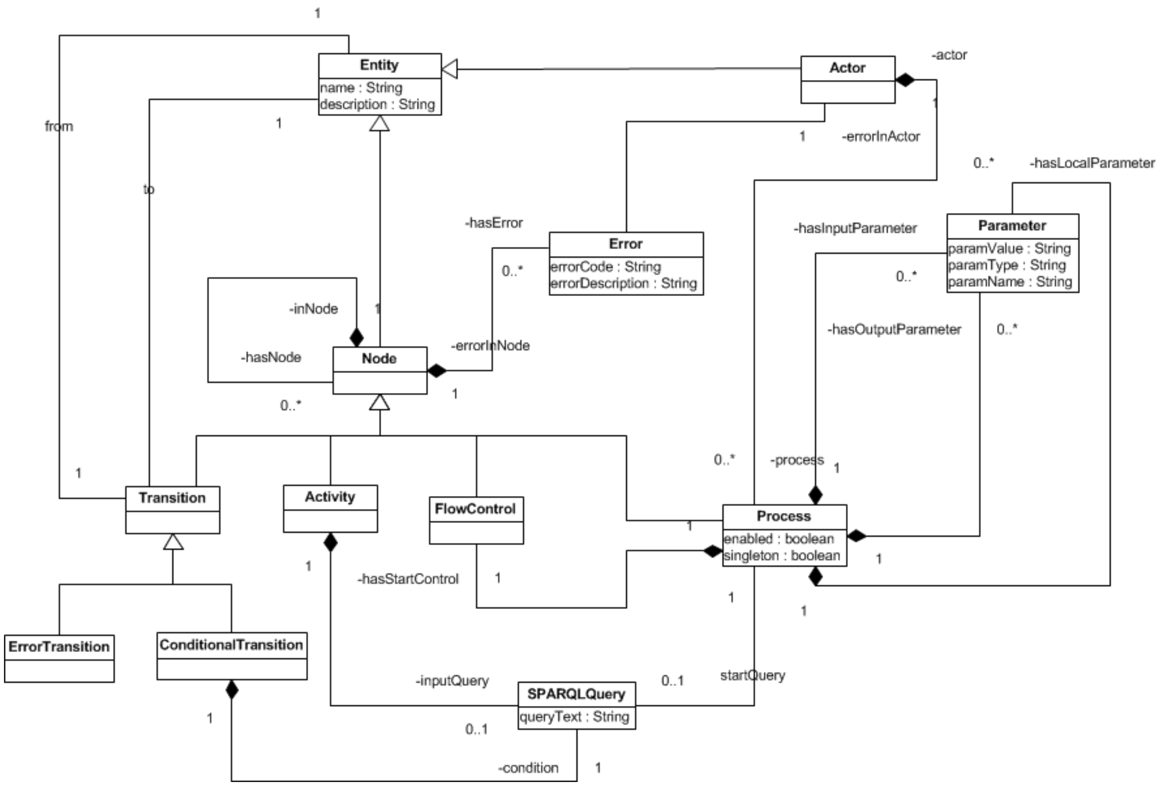


Figure 6.3: The process ontology class diagram. It shows all the classes used in this ontology. For readability purposes, the connections between them were simplified.

6.1.2.1 Entity

An entity is a basic structure that contains a name and a description. All the classes requiring both of these properties should be subclasses of *Entity*. This class is useful not only for the process ontology but also for all the ontologies that import it. This results in an extensive use of this class as a superclass of other concepts in many of the presented ontologies. The two properties defined for an Entity are:

- **name** - A String that contains the name of the entity.
- **description** - A String that contains a textual description of the entity to inform the users of anything related to it.

6.1.2.2 Node

This is the base class for all the functional components of the process ontology and a direct subclass of the *Entity* class. The use of a superclass like this can be justified by the fact that processes, activities, controls and transitions may be seen as nodes of a tree that can be inserted inside each other, this way parent and child nodes are allowed. It also helps in the graphical representation of a process, when using an editor to view it. The properties that define this class are:

- **hasNode** - Defines a sub-node of the current node. It is an inverse property of *inNode*. This is also a transitive property, which means that the sub-nodes of a given node are also sub-nodes of its parent node.
- **hasError** - If the node finished its execution with errors, they should be referenced using this property.
- **inNode** - Defines a parent node of the current node. It is an inverse property of *hasNode*.
- **nodePosition** - Binds the current node to an instance of *Coordinate*, which was described in the concepts ontology, that defines the position of the node in a graph. This property is useful when designing a graphical process editor or visualizer.
- **nodeDimension** - Binds the current node to an instance of *Dimension*, which was described in the concepts ontology, that defines the length and width of the node. Like in the previous property, this one is also useful for visualizing nodes in a graphical environment.

6.1.2.3 SPARQLQuery

It was necessary to insert whole SPARQL and SPARQL/Update queries inside process ontologies. The *SPARQLQuery* class was used to meet this need. The instances of this class can store queries in a String as well as variable bindings, much like expressions in OWL-S [W3C, 2004a].

This class makes possible the creation of conditions for transitions and input assignment in activities. The two properties defined for this class were:

- **expressionText** - This is a String that contains the query text in the SPARQL language format.
- **variableBinding** - A reference to a parameter representing of a variable defined inside the query text. This may be used to convert results from the SPARQL query into values that might actually be used in the processes. A *SPARQLQuery* may have more than one variable binding.

6.1.2.4 Parameter

This class is one of the most important classes in this ontology. Its purpose is to hold variable values in the processes. These variables may have pre-defined values or have their values assigned at run-time. This is useful to create bindings for queries, local variables or in other situations when the value or type of the variables is unknown before run-time. The parameter class is defined by the properties:

- **paramType** - The URI of the class of the value contained in the parameter. This URI may also point to datatype URIs like *String* or *Integer*.
- **paramName** - A String holding the name of the parameter. Useful for variable bindings or for informational purposes.
- **paramValue** - This property stores the actual value of the parameter. It may hold any type of value (Instances or native types).

6.1.2.5 Actor

An actor is an entity able to execute processes. Each actor has a set of processes bound to it, which means that these processes are only in the domain of their respective actor. Only the actor whose processes are defined as his own may execute them. In descriptions that extend this ontology where more detailed types of actors are needed, the *Actor* class should be extended by these new classes. This class only has one property defined:

- **process** - References a process that may be executed by the actor being defined. An actor may have more than one occurrences of this property.

6.1.2.6 Process

This is the class that defines an actual process. A process is composed of a series of activities that, connected with each other in a certain order, create a series of actions in a given domain. A process contains activities, transitions and controls as its sub nodes in order to define the flow of data and the ordering of activities. In this ontology, this class is a subclass of *Node*, since it may be displayed in a graph visualization tool. As for properties, the ones defined for a process are:

- **startQuery** - This property references an instance of *SPARQLQuery*. If the designer of the process requires it to be automatically executed in certain conditions, a *SPARQLQuery* instance with the corresponding query must be inserted in this property. So, if the query evaluates to *true* at any given moment, the process is executed. This may very well substitute the use of rules in many applications since it allows an if-then-else logic in the process engine. If the *SPARQLQuery* instance has variable bindings in it, these will be passed as inputs of the process.

- **hasStartControl** - The starting flow control must be defined so that an engine knows how to actually start running a process. This property points to the first flow control instance to be activated.

- **hasLocalParameter** - Processes may have local parameters which are only temporary values used when it is in an execution state. Local parameters are deleted when the process ends its execution. Other process engines already implement similar approaches [TIBCO, 2011b]. This property references an instance of the *Parameter* class.

- **hasInputParameter** - Input parameters may be defined for processes that need external values in order to be executed.

- **hasOutputParameter** - When a process is executed, it may output values for external entities to use.
- **enabled** - A boolean that states whether the process is enabled or not. This allows the process to be enabled/disabled at run-time for any reason necessary.
- **actor** - This is an inverse property of the *process* property in the *Actor* class. It references the actor instance that owns the process.
- **singleton** - A boolean value that, if true, marks the process as only being able to be executed once at a time. By nature, when a process needs to be executed, a new instance is created inside the engine. However, sometimes, certain processes must not have several instances being executed at the same time. This property allows the designer to control this situation.

6.1.2.7 FlowControl

As stated in Chapter 5, processes need a way to control the flow of the executed activities (Whether they should be executed sequentially, in parallel, etc.). The purpose of this class is to give a template for flow control description. Actual flow controls are subclasses of this class. All the properties described in the following itemization are sub-properties of the *hasNode* property:

- **hasActivity** - References instances of the class *Activity* contained inside a flow control. This allows the engine to know which activities to load for any given control.
- **hasControl** - Controls can have inner controls in order to create processes with more complex behaviours. This property references all the sub controls.
- **hasTransition** - Controls also contain information about the transitions between their inner nodes. This is because transitions are treated as nodes, helping with the optimization of process loading.
- **hasProcess** - A process may need to invoke another process defined in the same actor. This property allows controls to place sub-processes inside them.

As previously stated, the *FlowControl* class is just an abstract class. The actual flow control implementations are shown in Figure 6.4 and described afterwards.

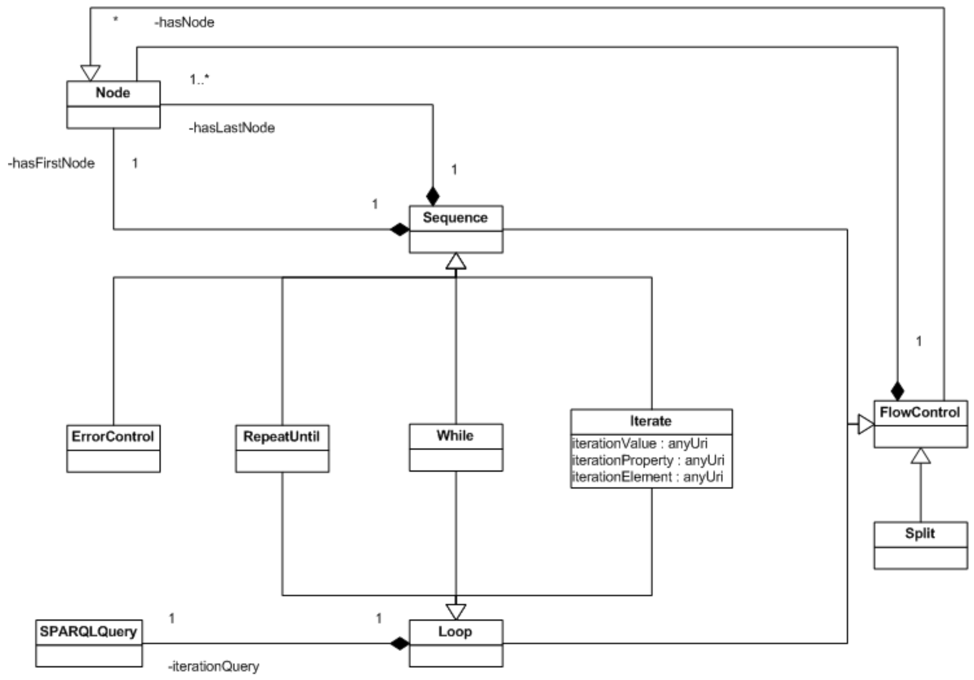


Figure 6.4: The flow control class diagram, showing the created classes for process control.

- **Sequence** - This subclass of the flow control class states that any activity, control or process contained inside must be executed sequentially. Even if there are two Transitions that evaluate to *true* from one activity, only one will be selected. In order for the engine to know how this control starts and when this control ends, the properties *hasFirstNode* and *hasLastNode* where defined pointing to the first node to be executed and to the last node to be executed, respectively. It should be clear that a sequence may have more than one final node to allow branching.

- **Split** - Is a flow control that allows the nodes contained in it to be executed in parallel. Its execution finishes when all the children nodes have finished theirs. No transitions are allowed in a split control, so it is best used in coordination with other controls to achieve a more complex behaviour. It has no specific properties.

- **Unordered** - All the nodes contained in this control are executed in no specific order. It depends only in the order they were read from the process definition. Like in the split control, no transitions are allowed inside the unordered control.

- **Loop** - This is an abstract subclass of the *FlowControl* class. This class is a template class that allows the nodes contained in it to be executed repeatedly until a certain condition is evaluated to *true*. Only one property was created for this class, the *iterationQuery*, which references an instance of a *SPARQLQuery* that contains the query evaluated for the loop. All the subclasses of this control are also subclasses of the sequence control and they are:
  - **Iterate** - Reads all occurrences of a given property filled for a given instance and iterates through them until there are no more occurrences to be read. The iterate control has thee properties, the *iterationElement*, where the instance whose property is to be read is defined, the *iterationProperty*, containing the URI of the property to iterate and the *iterationValue*, that will be filled at run-time with the current value to be analysed.
  - **RepeatUntil** - Repeats the execution of all the nodes contained in it until the *iterationQuery* evaluates to *true*.
  - **While** - This control loops over the contained nodes while the *iterationQuery* is *true*.
- **ErrorControl** - This special control class was created for error handling as defined in Chapter 5.2. It has no properties defined.

6.1.2.8 Activity

Activities are the atomic actions a given process may execute. This class by itself, when instantiated, will not create valid activities, since it is just a template definition. A developer must subclass it in order to create more fine-grained activities. Every activity may have inputs, outputs and even configuration values. It is a subclass of the *Node* class, since it may be represented as a graph node. The defined properties for this class are:

- **inControl** - Defines the parent flow control of this activity and is a sub-property of *inNode*.
- **inputQuery** - Input values, in a given process, are passed between activities by means of a *SPARQLQuery*. Languages such as SPARQL can be used to query the data model of the current process execution for outputs of previous activities or local

parameters and assign the desired values to the inputs of an activity. This property serves this purpose by pointing to a query to assign the values as inputs of this activity.

When creating a subclass of *Activity*, the designer also needs to provide certain fixed values for the new activity. These values come in the form of OWL annotation properties, which are constant throughout the instances of a class. In this context, the required annotation properties are:

- **configuration** - States the Java instance that should be used to configure an instance of this activity in case an editor is being used to create processes.
- **icon** - Provides the path of an icon associated with the new activity type. An icon is useful for visual representations of processes.
- **implementation** - References the Java instance responsible for the actual implementation of the behaviour of the newly created type of activity.
- **label** - This is a generic OWL annotation property. In this case it is used to place a textual representation of the new type of activity. Useful for both editing and visualizing processes.

6.1.2.9 Transition

Along with flow controls, transitions help defining the flow of the process. They point to the next node to execute and may help with optional executions. Transitions are also useful for a designer to understand the flow of a given process. Since a transition may exist inside other nodes, it was created as a subclass of the *Node* class. The defined properties for transitions are:

- **from** - References the instance of the previous node in the transition.
- **to** - References the instance of the next node in the transition.

6.1.2.10 ConditionalTransition

In order to have branching in the execution of a process, there is a subclass of the *Transition* class called *ConditionalTransition*. This type of transitions are only executed when a certain condition evaluates to *true*. The only property defined for this class is:

- **condition** - When executing a process, the designer may want it to flow in a certain path when a given condition is *true* and flow in another path when this condition is *false*. For this purpose, any conditional transition may reference a *SPARQLQuery* that is evaluated when the transition is reached. If this condition is *true*, the next node in the transition is executed, if not, the transition will not allow it to be executed.

6.1.2.11 ErrorTransition

When a node finishes its execution with errors and if it has an error transition coming from it, an alternate path to recover from that error may be created, as described in Chapter 5.2. This class is a subclass of the *Transition* class and has no properties defined for it.

6.1.2.12 Error

An instance of this class should be used whenever a node finishes execution with errors. This class describes the error and can be subclassed to create more fine-grained error definitions. Three properties were defined for instances of this class:

- **errorActor** - References the actor in which the error occurred.
- **errorInNode** - References the node where the error occurred.
- **errorID** - A String containing a text identification of the error in order for further analysis of it.
- **errorDescription** - A String containing the description of the error, suitable for a user to be able to have a more detailed information of what happened.

6.1.3 Agent Ontology

Since the process engines work in a MAS, an ontology was required to define agents and their properties, so that queries and message exchange could be performed in OWL. This ontology defines classes compatible with the ones defined in the JADE framework and its URI is http://www.fct.unl.pt/ontologies/eas-agent-ontology.owl. Also, to be compatible with process execution, this ontology imports the classes from the process domain. In Figure 6.5, the UML class diagram of this description is shown.

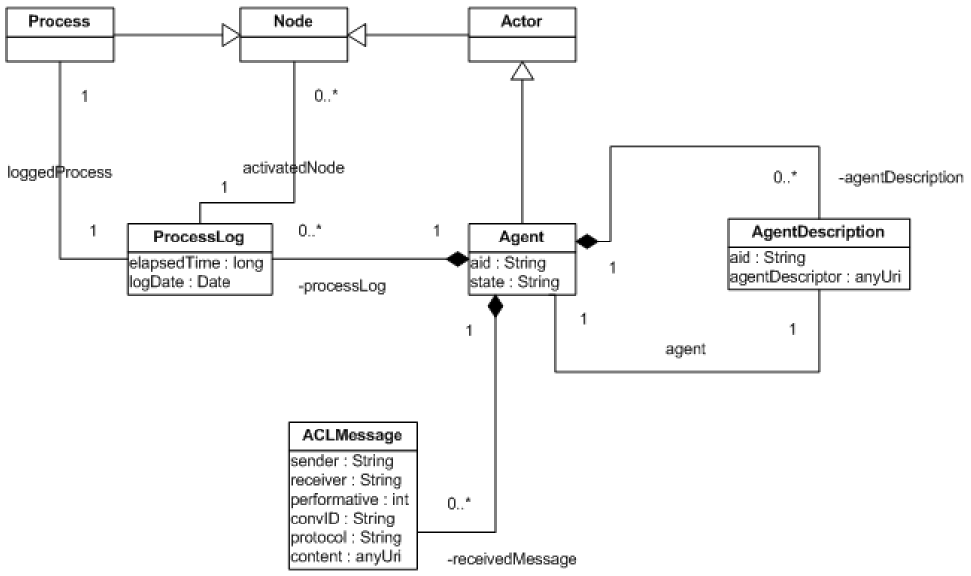


Figure 6.5: The agent ontology class diagram. This UML diagram shows the classes defined for the agent ontology in order to provide compatibility with the JADE framework.

In the figure, some classes of the process ontology are shown in order to best understand the relationships between that ontology and the one being currently described. In the following sub-sections the classes in this ontology are described.

6.1.3.1 Agent

As an agent executes processes, the *Agent* class was defined as a subclass of *Actor*. This class defines the basic information of a JADE agent in the system. The defined properties are:

- **aid** - A String that contains the textual representation of the Agent Identifier (AID) of the agent.
- **isDescribedBy** - Holds the instance that describes the agent. This property was created to support the extension of the system. As an example, in the case of this thesis, this property will reference to the instance that describes the module the agent represents.
- **receivedMessage** - This property references instances of the *ACLMessage* class, described in the next section, that represent messages received by the agent.

- **state** - Holds a String describing the current state of the agent. In this implementation, this property only supports three values: *SETUP*, *RUNNING* and *END*.
- **agentDescription** - References instances of the *AgentDescription* class. These instances represent neighbour agents.
- **processLog** - References instances of the *ProcessLog* class, which contain the logs of the executed processes.

6.1.3.2 ACLMessage

In order to give this system some compatibility with the JADE framework, this class was created to contain the descriptions of the ACL messages. This class allows the agents to insert the descriptions of the exchanged messages in their KB in order to be able to execute queries about them as with any other description they may contain. Therefore, the properties of this class mimic the properties of its sister class in JADE:

- **content** - This property references the main instance of the content of a given message. Since a message can be sent using any class, this property supports all the possible classes.
- **convID** - A String holding the conversation identification of the message, useful for identifying messages within exchanges using the same protocol.
- **performative** - An integer that represents the performative of the message. In JADE, performatives are what distinguish the message types. As examples, performatives may state that a certain message is of the *Request* or of the *Inform* types.
- **protocol** - A String representing the name of the protocol the message belongs to. Protocols may be FIPA Interaction Protocols or any other protocol the designer may decide to use.
- **receiver** - The String representation of the AID of the receiver of the message, which is useful when compiling a certain message to be sent.
- **sender** - The String representation of the AID of the sender of the message, which allows the receiver to respond to messages.

6.1.3.3 ProcessLog

An useful feature of the agents in this system is keeping logs of the processes that were executed, as well as which nodes were activated during the respective execution. A *ProcessLog* instance keeps these logs by filling the properties below:

- **activatedNode** - References an instance of a node that was activated during the execution of the process.
- **logDate** - Holds the date of execution of the process, allowing the designer to know exactly when a given instance of a process was created.
- **elapsedTime** - A long containing the total milliseconds the process took to finish execution.
- **loggedProcess** - References the instance of the logged process.

6.1.3.4 AgentDescription

Agents need to have short descriptions of the other agents in the network in order to be able to query or alter their KBs. The currently needed information is stored in the following properties:

- **aid** - A String containing the text representation of the AID of the remote agent, so that agents are able to send queries between them using the JADE messaging system.
- **agent** - References the instance of the *Agent* class of the remote agent, which helps identifying the remote agent in queries.
- **agentDesciptor** - References the main instance of the description of the remote agent. This is just an utility reference to make queries easier and smaller.

6.1.4 Communication Ontology

Communication between agents may be performed using OWL instead of the built-in ontology format of JADE. With this in mind, an ontological specification was created for the communications. This ontology is used for direct communication between the agents and specifies the actions or other information the agents may share in their message exchanges. This ontology imports the agent ontology to be aware of its concepts during message exchange and its URI is http://www.fct.unl.pt/ontologies/eas-comm-ontology.owl. In Figure 6.6 an UML class diagram is displayed for better comprehension.

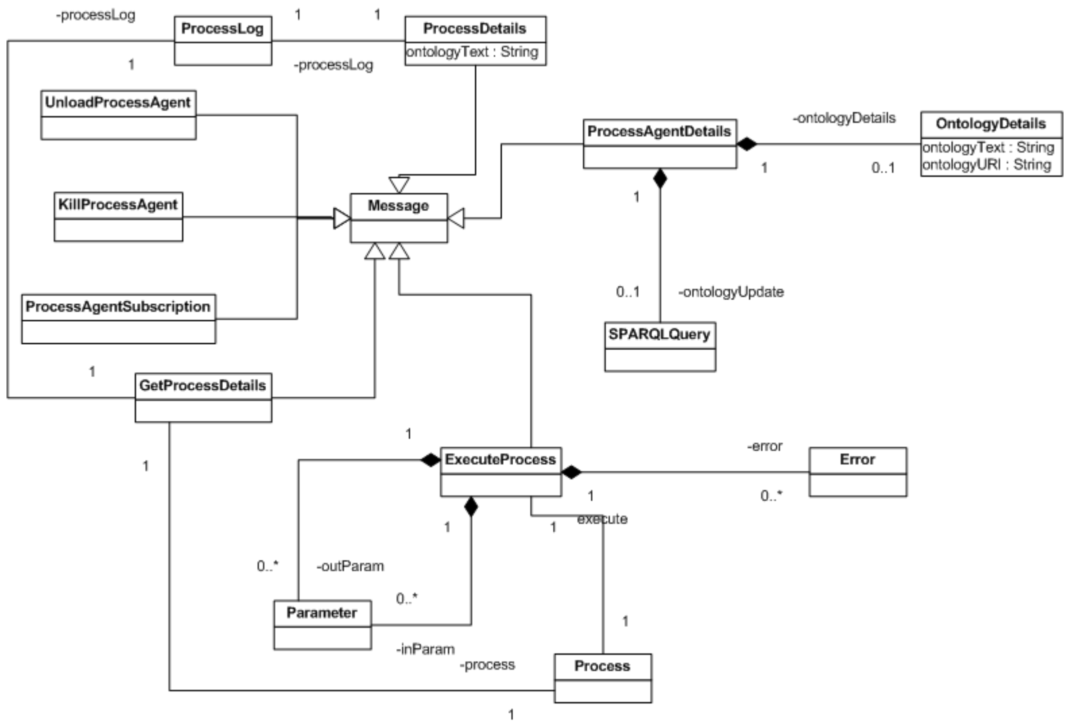


Figure 6.6: The communication ontology class diagram, which contains the default classes for agent communication in this system.

As with the previous diagrams, classes from imported ontologies were placed in this diagram for better comprehension of the relationships between the concepts.

6.1.4.1 Message

This is the base class for all the classes in the message ontology. All other classes in the communication domain that represent communicative acts, are subclasses of *Message*.

6.1.4.2 ProcessAgentSubscription

A message containing an instance of this class is intended to create a subscription in the target Process Agent to get details about its ontology.

6.1.4.3 ProcessAgentDetails

When a Process Agent receives a subscription message, the agent sends its details, using instances of this class. It may contain the full ontology of the agent in a textual form or simple SPARQL/Update queries with the latest updates in the KB of the agent. An instance of this class may have two properties:

- **ontologyDetails** - References an instance of the *OntologyDetails* class, which contains the full textual description of the ontology in the agent.
- **ontologyUpdate** - References the instance of a *SPARQLQuery* containing the latest KB updates of the agent.

6.1.4.4 OntologyDetails

An instance of this class contains the full text description of an ontology as well as the corresponding URI. This is useful for agents to send their full KB contents to other agents. This class has two properties:

- **ontologyText** - A String with the full text description of a given ontology.
- **ontologyURI** - A String with the URI of the described ontology.

6.1.4.5 ExecuteProcess

Process Agents can be requested to execute a given process. When this is the case the initiator agent must send a message with an instance of this class. Three properties were defined for an instance of the class *ExecuteProcess*:

- **execute** - References the instance of the process to execute, since Process Agents need to search correct process to execute it.
- **inParam** - References the *Parameter* instances with the correct values the process has as inputs.
- **outParam** - After the process is executed and if it returns any parameters, the values of these are put in this property when the information message is sent.

- **error** - Processes may finish their execution with errors. If that is the case, these must be sent to the requester agent in order for it to analyse them. This property makes sure that the proper errors are referenced in the message.

6.1.4.6 UnloadProcessAgent

When a Process Agent receives a request containing an instance of this class, it should perform a clean shut down, by finishing execution of all the processes and checking if there are any final processes to execute.

6.1.4.7 KillProcessAgent

A Process Agent can be requested to terminate its execution immediately without further checks. An instance of this class in a request makes sure of this.

6.1.4.8 GetProcessDetails

Processes, whenever executed, store the execution information in a private KB inside the Process Agent, apart from the other KBs. This is why the Monitor Agent does not have direct access to this information and has to explicitly make a request to retrieve it. When the Monitor Agent needs to get the details of a specific logged process execution, it sends a message containing an instance of this class. Two properties were defined for the class *GetProcessDetails*:

- **process** - Specifies from which process the logged execution should be retrieved. This property is used for faster search inside the Process Agents.

- **processLog** - References the instance of the process log of the previously specified process. This log should be searched within the specified process by the receiving agent.

6.1.4.9 ProcessDetails

When a Process Agent receives a request from another agent to get the private details of a specific execution, it bundles that information in an instance of this class and then sends it in the reply message. For the *ProcessDetails* two properties were specified to store the information:

- **processLog** - References the instance of the process log of whose details are being sent. This is for the receiver to know how to process the requested information.
- **ontologyText** - Contains a String with the textual description of the ontology contained in the requested log. This description contains all the information stored during a specific execution of a process.

6.2 The Implemented MAS

In Chapter 4 it was explained that the generic MAS is composed of two types of agents: the Process Agent and the Monitor Agent. These agents were implemented using JADE, while their KBs were programmed using Jena ontology models. Both of these technologies were already explained in Chapter 3.

Messages in this system are exchanged in OWL, SPARQL or SPARQL/Update languages. In all of the cases, the chosen approach was similar to the one already used in a project called *Agent OWL*, in which several agents exchange messages in String format using these languages [Laclavık et al., 2006]. As described in Section 3.1.3, FIPA-ACL messages have certain useful parameters that may be filled in a message so that the receiving agent may understand its contents and, among these is the *language* parameter. For agents to correctly act according to the received message, the *language* parameter must be filled with the correct language used in the content (OWL, SPARQL or SPARQL/Update). If not, the message is discarded.

Normally, a SPARQL message is a query to the KB inside an agent, which will require a response containing the results, depending on the operation specified. SPARQL operations were already covered in Section 3.2.3. This behaviour is ideal for the FIPA Query Interaction Protocol.

A SPARQL/Update message implies an update to the KB inside a given agent. In this system these messages also use the FIPA Query Interaction Protocol, although the response message has no content. It only states whether the update was successful or not. The update not being successful, might mean that the query has an incorrect syntax.

Consequently, queries in SPARQL or SPARQL/Update are sent between agents in the ACL message format shown in Listing 6.1.

Listing 6.1: An example of an ACL message containing a SPARQL query. In this example, the language property is set to *SPARQL* and the query is located in the content of the message.

```
(QUERY-REF
 :sender  ( agent-identifier :name agent1@10.176.233.182:1099/JADE
                 :addresses (sequence http://example.com:7778/acc ))
 :receiver  (set ( agent-identifier :name agent2@10.176.233.182:1099/JADE ) )
 :language  SPARQL
 :protocol  Process-Execution
 :conversation-id  ID_17970505_3835313490176
 :content
        SELECT  ?s
        WHERE
        {
        ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                <http://www.fct.unl.pt/ontologies/eas-system-ontology.owl#Unit> .
        }
  )
```

Messages in OWL are a special case. This language is used for agents to trade information that has to be analysed and acted upon. While SPARQL and SPARQL/Update messages are merely orders to be directly used inside a KB, OWL messages must be interpreted. As far as the system knows, messages in this language are only small fragments of KBs that are being exchanged between agents. In technical terms, to send a message, the agents create a new, independent KB, via a Jena ontology model, and convert it to String, much like in *Agent OWL*. The implementation of this method can be seen in Listing 6.2.

Listing 6.2: Java code to convert Jena ontology models to String. In this listing, the method receives a model, whose contents are written to a String.

```
        public static String modelToString(Model model)
        {
                StringWriter writer = new StringWriter();
                model.write(writer, "RDF/XML-ABBREV");
                return writer.toString();
        }
```

When received, messages need to be converted back to a Jena ontology model for an agent to be able to analyse it and do whatever it needs to do with it. The code for this conversion can be seen in Listing 6.3.

Listing 6.3: Java code to convert ontology String representations to Jena ontology models. This method receives a String containing an ontology and parses it to a new Jena model.

```
public static Model stringToModel(String text)
{
        StringReader reader = new StringReader(text);
        Model model = ModelFactory.createDefaultModel();
        model.read(reader, null);
        reader.close();
        return model;
}
```

Messages in OWL are composed of a main instance, which contains the core information of their purpose. For example, a message that orders an agent to kill itself, contains a main instance of the class *KillProcessAgent*. But this main instance may refer to other instances contained inside the message and so on, which, adding to the fact that neither JADE nor Jena have a way of determining which instance is the main instance in the contents of any given message, creates a problem in analysing the message.

As described in Section 3.1.3, the FIPA-ACL standard provides a way of adding user-defined parameters to the ACL messages. To cope with the problem of searching for the main instances of a message, two new parameters were defined for any exchange of messages in the OWL language:

- **X-MAIN** - Contains the URI of the main instance in the contents of a message. This is how usually an agent finds this instance in order to know what the subject of the message is.
- **X-TYPE** - Contains the URI of the type of the main instance of the message. This field was not required for agents to analyse a message but adds to the performance. If this field is wrong or points to a wrong ontological class, the message may be discarded without having to convert it to a Jena ontology model.

In the current system, unless the message is sent in the SPARQL language, the described fields are obligatory. This way agents may search for the main instance inside a message to get the needed information. An example of and ACL message with the content in OWL is shown in Listing 6.4.

Listing 6.4: An example of an ACL message containing an OWL content. One should not how the parameters *X-MAIN* and *X-TYPE* point to the main instance and its respective type.

```
(REQUEST
 :sender  ( agent-identifier :name agent1@10.176.233.182:1099/JADE
                :addresses (sequence http://example.com:7778/acc ))
 :receiver  (set ( agent-identifier :name agent2@10.176.233.182:1099/JADE ) )
 :language  OWL
 :protocol  Process-Execution
 :conversation-id  ID_17970505_3835313490176
 :X-MAIN http://example/agent1.owl#execute
 :X-TYPE http://www.fct.unl.pt/ontologies/eas-comm-ontology.owl#ExecuteProcess
 :content
        <rdf:RDF
        xmlns:proc="http://www.fct.unl.pt/ontologies/eas-process-ontology.owl#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns:comm="http://www.fct.unl.pt/ontologies/eas-comm-ontology.owl#">

        <comm:ExecuteProcess rdf:about="http://ex/agent1.owl#execute">
                <comm:execute>
                        <proc:Process rdf:about="http://ex/agent2.owl#agent2_proc"/>
                </comm:execute>
        </comm:ExecuteProcess>
        </rdf:RDF>
  )
```

One might have noticed that the name in the *protocol* field is neither one of the already discussed FIPA protocols in both Listing 6.1 and Listing 6.4. To provide extension to the system, every exchange of messages already pre-defined, like a process execution request, contains the *Process-Execution* name in the protocol field. This allows the users to use the correct names for FIPA Interaction Protocols if needed.

6.2.1 The Process Agent

In the JADE framework, all the agent implementations must extend the generic class *Agent*. In the context of this work, an intermediate class was created, *OntAgent*, which extends *Agent*. This class provides generic methods for printing identified messages in the console, registering the agent in the JADE DF and loading ontologies in OWL by means of the Jena framework. The Process Agent is an extension of *OntAgent*, inheriting all its basic functionalities and adding a process engine as well as other utility methods.

A Process Agent must be initialized with three specific arguments or it will shut itself down before performing any other action:

- The path of the file containing the ontological description the agent needs to load to its KB. This file must contain both the description of the entity the agent represents and the definitions of the processes it executes.
- The URI of the loaded ontology. This is needed because Jena uses it to create several enhancements in the KB for enhanced writing of the ontology.
- The URI of the *Agent* class instance the agent has to represent in order for the it to gain an identity of its own.

During the initialization the agent goes through the steps described in Figure 6.7 before it can be fully functional. In some of these steps, if they fail, the agent will shut itself down.

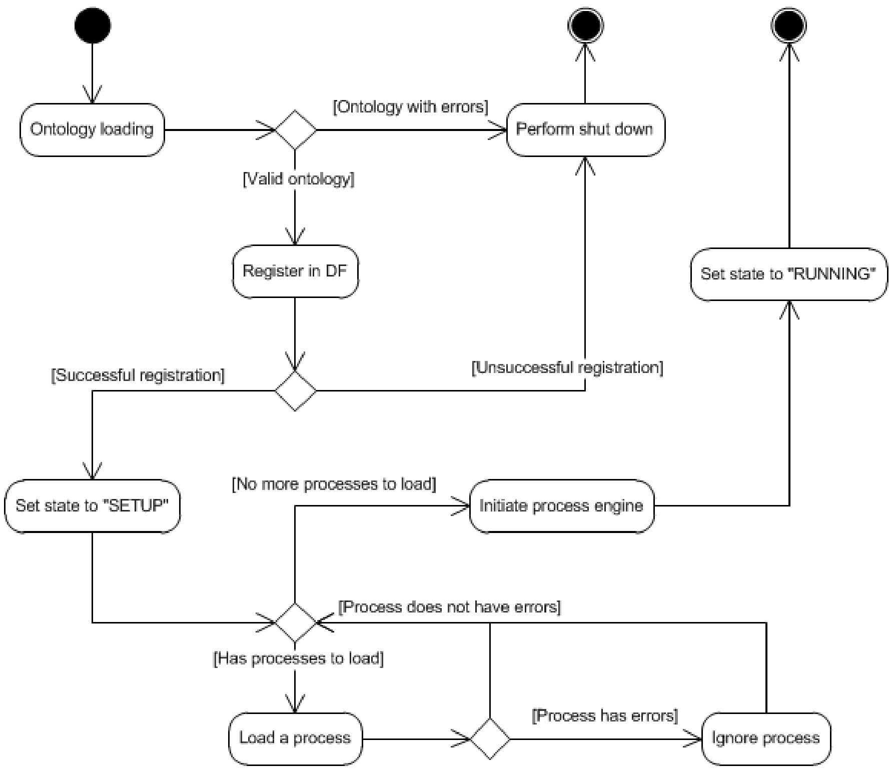


Figure 6.7: The initiation of the Process Agent. This UML state diagram shows all the steps the Process Agent goes through in order to initiate.

There are two types of finalization the Process Agent can execute upon itself when requested. The first type is a *clean* finalization, where the agent still finalizes its process engine in order to properly finish process execution and is shown in Figure 6.8. The second type bypasses the engine finalization and simply takes the agent down.

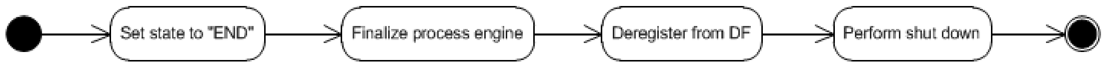


Figure 6.8: The finalization of the Process Agent. The steps shown in this UML state diagram are executes whenever the Process Agent performs a *clean* shut down.

In Section 4.1 it was mentioned that Process Agents could access each others KB in order to share knowledge. To make this possible, a Process Agent is equipped with two Jena ontology models: one is used for the agent to keep his own ontology in and the other to make possible the querying external ontologies as if it was the agent's own ontology. Also, each agent has what was called of a *Proxy Agent*, which is a mirror agent that contains exactly the same ontology as the original Process Agent. This Proxy Agent is updated whenever the corresponding Process Agent is updated. These are the base assumptions to get the agents to share their ontologies with the community.

When a Process Agent becomes active, it registers itself in the default JADE DF agent, putting the following information in the registration entry:

- The URI of the *Agent* class. This is the identity of the agent, which was passed as an argument at the beginning its initialization.
- The URI of the corresponding agent description, defined by the *isDescribedBy* property. This identifies what entity is defining the agent.
- The URI of the ontology the agent has just loaded. The ontology the agent loaded is its own domain, so this informs the other agents who is taking care of what.

Agents also subscribe to the default DF service allowing them to receive a message whenever an agent has registered itself in it. So, whenever a new registration message arrives, the agents store the previously described details in their own KB by means of an instance of *AgentDescription*. They also create a new ontology model and add it as a sub model of the shared ontology model as can be seen in Figure 6.9.

The ontology models for each of the external agents contain a graph created by the author, which will not contain any information stored inside. Instead, these graphs route

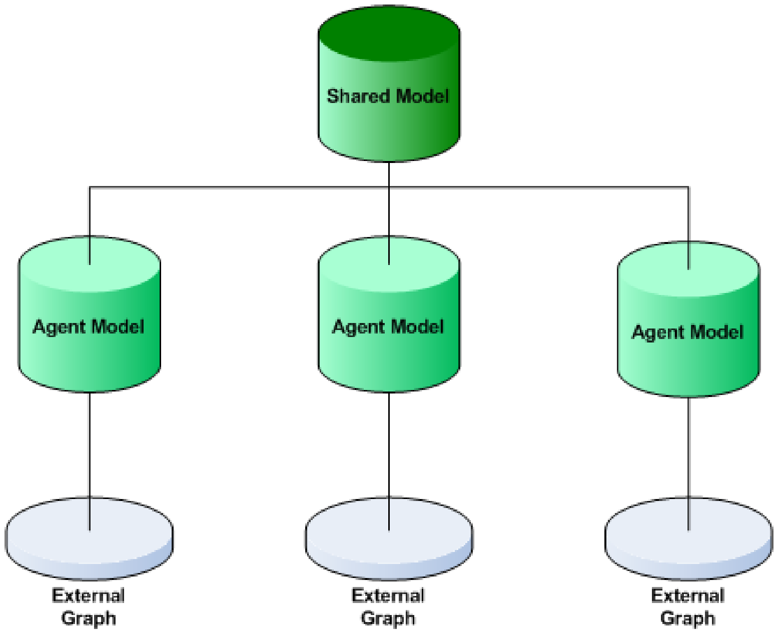


Figure 6.9: The shared model tree. This tree shows that a shared model exists containing a sub-model for each of the other agents in the network. Whenever an operation is made one of the models, the external graph routes it to the corresponding agent.

the queries and assertions to the corresponding agent, which will perform them in its default KB model. This way the agents do not need to store the full KB information of the external agents, while remaining able to query them as if querying one of their own KBs.

The above explanation still does not explain the reason for Proxy Agents. When querying external KBs, the agents had to do so without recurring to behaviours, since the Jena framework is not prepared to work directly with the JADE framework. Due to this fact, while a Process Agent was querying another agent, its current thread of execution would be blocked until the query was completed. It was already referred in Section 3.1.4 that JADE agents have only a single thread of execution. The problem with this approach was that, if two agents queried each other at the same time, both would be blocked, which meant that no one would respond to the query rendering both agents useless. To resolve this conflict, the agents, instead of querying each other, they query their corresponding Proxy Agents which will never be blocked. This workaround was the same as having two threads running in the same agent but, since the JADE framework is not directly prepared to handle two threads receiving and sending messages from the same agent, the Proxy

Agent solution was used. A two-thread solution would be viable if the JADE framework was extended for that purpose, which would consume much programming time and certain architectural changes in the current system. As for assertions to the KB in the agents, there is no problem if they are directly made in the Process Agent instead of the Proxy Agent. In Figure 6.10 the interactions between the Process Agents and the Proxy Agents are shown.

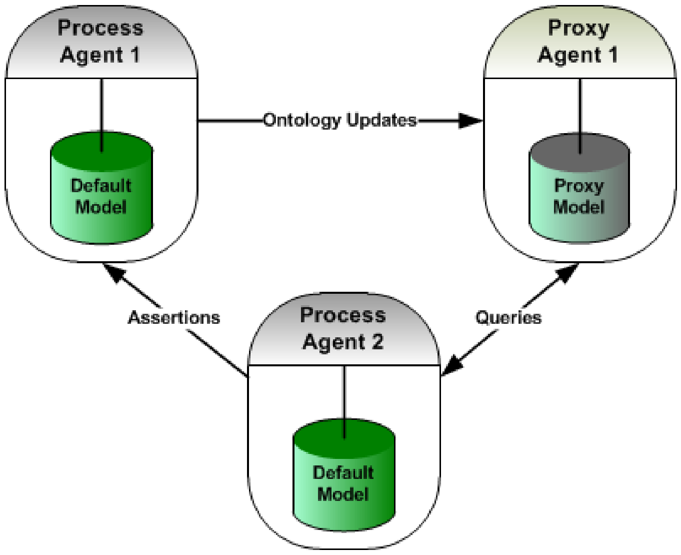


Figure 6.10: Interactions between Process Agents and Proxy Agents. The Process Agents constantly update their corresponding Proxy Agents so that queries the queries always return updated values. Assertions are made directly in the Process Agents.

The configuration described above, plus the ability of Jena to create common model stores to query models for information, allows the agent to query external agents transparently as if it is querying its own KB. Which simplifies the work and keeps all the ontological descriptions in the system coherent, since updates in a certain domain may only be done in the corresponding agent.

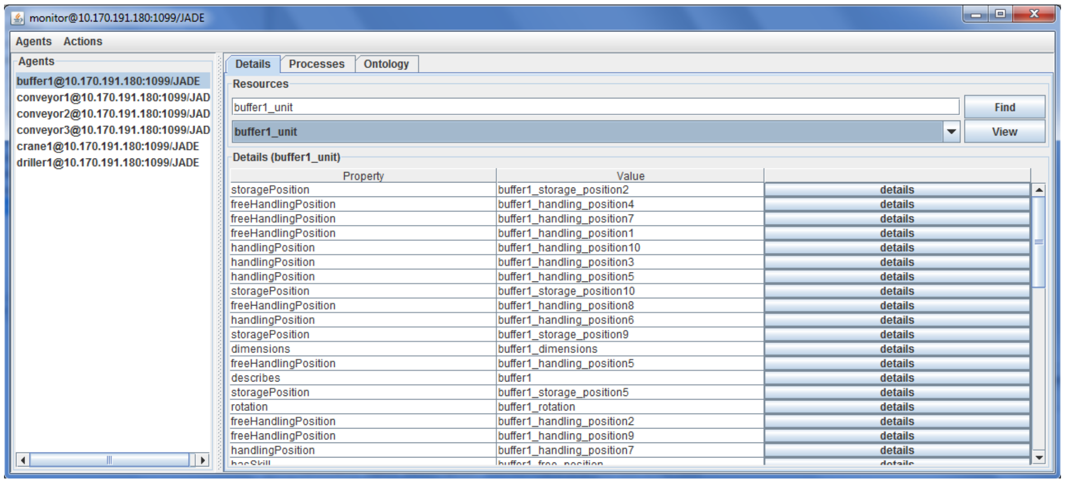
As for active behaviours, the Process Agent has one that is always listening for requests, whether they are to execute a process or to shut the agent down. Another behaviour listens for subscriptions for the ontological information in the agent and sends an information message whenever its description has been altered. The Process Agent is also equipped with a behaviour that receives any type of messages and inserts them in its KB, by creating and filling an instance of the *ACLMessage* ontology class.

As for the process engine, since it is one of the main objectives of this work, is described in Section 6.3.

6.2.2 The Monitor Agent

The Monitor Agent serves as a generic GUI so that the designer may monitor and analyse the operating Process Agents and the data they contain. Although the Process Agent can be activated independently, the Monitor Agent allows for the user to create agents and attach them to the corresponding ontologies.

Figure 6.11 shows the basic set-up of the GUI associated to the Monitor Agent, containing several useful features do diagnose the system.



Agents

buffer1@10.170.191.180:1099/JADE
conveyor1@10.170.191.180:1099/JAD
conveyor2@10.170.191.180:1099/JAD
conveyor3@10.170.191.180:1099/JAD
crane1@10.170.191.180:1099/JADE
driller1@10.170.191.180:1099/JADE

Resources

buffer1_unit

buffer1_unit

Details (buffer1_unit)

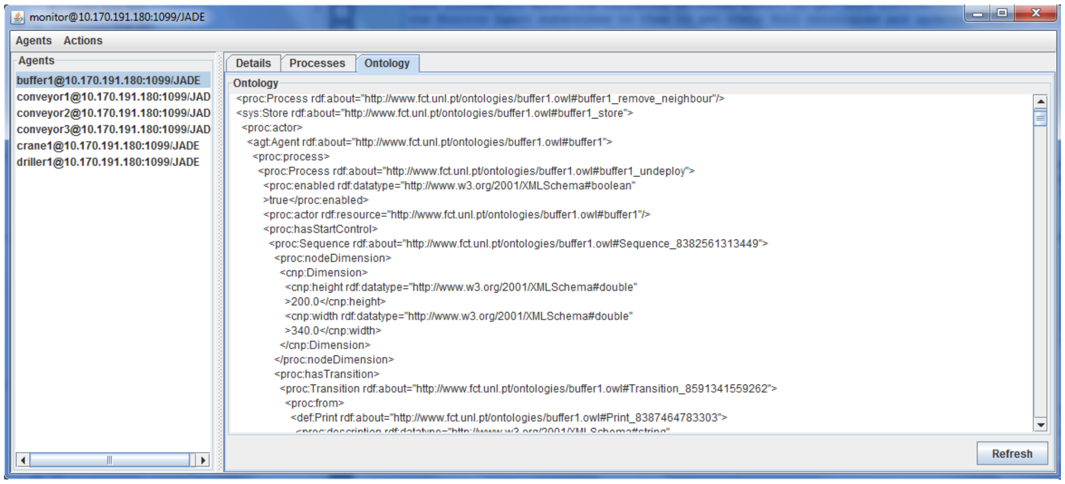
Property	Value	
storagePosition	buffer1_storage_position2	details
freeHandlingPosition	buffer1_handling_position4	details
freeHandlingPosition	buffer1_handling_position7	details
freeHandlingPosition	buffer1_handling_position1	details
handlingPosition	buffer1_handling_position10	details
handlingPosition	buffer1_handling_position3	details
handlingPosition	buffer1_handling_position5	details
storagePosition	buffer1_storage_position10	details
freeHandlingPosition	buffer1_handling_position8	details
handlingPosition	buffer1_handling_position6	details
storagePosition	buffer1_storage_position9	details
dimensions	buffer1_dimensions	details
freeHandlingPosition	buffer1_handling_position5	details
describes	buffer1	details
storagePosition	buffer1_storage_position5	details
rotation	buffer1_rotation	details
freeHandlingPosition	buffer1_handling_position2	details
freeHandlingPosition	buffer1_handling_position9	details
handlingPosition	buffer1_handling_position7	details
hasSkill	buffer1_free_position	details

Figure 6.11: The GUI associated to the Monitor Agent. This image also shows the *Details* tab, which provides details related to the ontologies contained in the selected Process Agent.

In the figure, to the left a list of active Process Agents can be seen. This agent has a subscription in the DF agent that informs it whenever a Process Agent was activated or deactivated. In the center several tabs show information about the currently selected agent. To get this information from the agents, the Monitor Agent subscribes to them to get their full ontologies and get updates whenever these ontologies are modified. On the top, a menu bar provides several options for the user to interact with the system.

Also, the figure shows one of the informational tabs this agent offers. This tab shows the details about a given instance in the ontology of the selected agent, by inserting all its statements in a table for easier reading. This tab also allows to search for an instance to get the details from or directly selecting it using a drop-down list.

If the table view is not enough to analyse the ontologies, their a textual representations can be accessed using the *Ontology* tab. See Figure 6.12.


monitor@10.170.191.180:1099/JADE
Agents
Actions
Agents
buffer1@10.170.191.180:1099/JADE
conveyor1@10.170.191.180:1099/JAD
conveyor2@10.170.191.180:1099/JAD
conveyor3@10.170.191.180:1099/JAD
crane1@10.170.191.180:1099/JADE
driller1@10.170.191.180:1099/JADE
Details
Processes
Ontology
Ontology
Refresh

```
<proc:Process rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1_remove_neighbour"/>
<sys:Store rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1_store">
 <proc:actor>
  <agt:Agent rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1">
   <proc:process>
    <proc:Process rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1_undeploy">
     <proc:enabled rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
     >true</proc:enabled>
     <proc:actor rdf:resource="http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1"/>
     <proc:hasStartControl>
      <proc:Sequence rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#Sequence_8382561313449">
       <proc:nodeDimension>
        <cnp:Dimension>
         <cnp:height rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
         >200.0</cnp:height>
         <cnp:width rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
         >340.0</cnp:width>
        </cnp:Dimension>
       </proc:nodeDimension>
       <proc:hasTransition>
        <proc:Transition rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#Transition_8591341559262">
         <proc:from>
          <def:Print rdf:about="http://www.fct.unl.pt/ontologies/buffer1.owl#Print_8387464783303">
```

Figure 6.12: The *Ontology* tab. This tab provides a textual representation of an ontology.

As was previously mentioned, the primary role of Process Agents is managing and executing processes. Therefore, this agent also provides a way to monitor the loaded and executed processes of a Process Agent. This monitoring is done in the *Processes* tab of the GUI, shown in Figure 6.13.

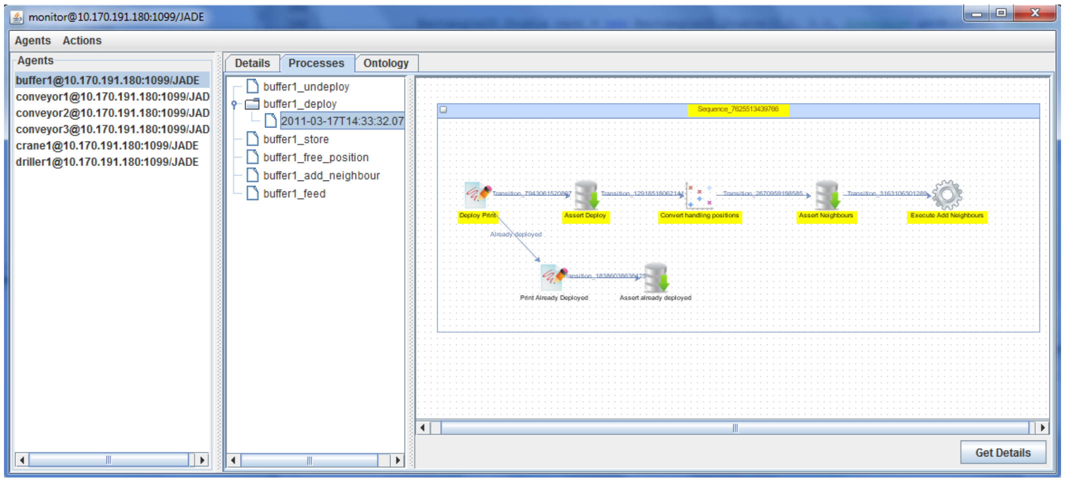
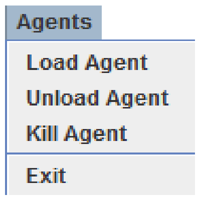


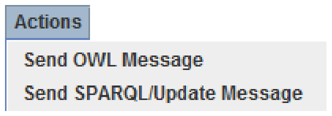
Figure 6.13: The *Processes* tab. This tab provides details about processes contained in Process Agent. It also shows which nodes were activated upon execution.

The tab in the figure shows the details of the processes the selected agent contains. On the left side, there is a tree that shows the loaded processes as well as their logged executions ordered by date. When any of these items in the tree is selected, the corresponding process graph is shown in the center of the tab. If a log was selected, the graph will highlight the activated nodes during the logged execution. Below, there is a *Get Details* button that, when clicked, will request the selected agent for details of the logged execution, which will, in turn, be shown in a new dialog. These details consist on the contents of the private KB of the executed process. This private KB will be discussed in Section 6.3.

The GUI in the Monitor Agent also contains a menu bar that allows the user to perform certain operations in the system. This menu bar contains the two menus shown in Figure 6.14.



(a) Agent Menu.



(b) Actions Menu.

Figure 6.14: The menus in the GUI of the Monitor Agent.

This agent is able to load, unload or kill Process Agents. All three actions can be achieved by accessing the *Agent* menu in the bar, whose details are shown in Figure 6.14a.

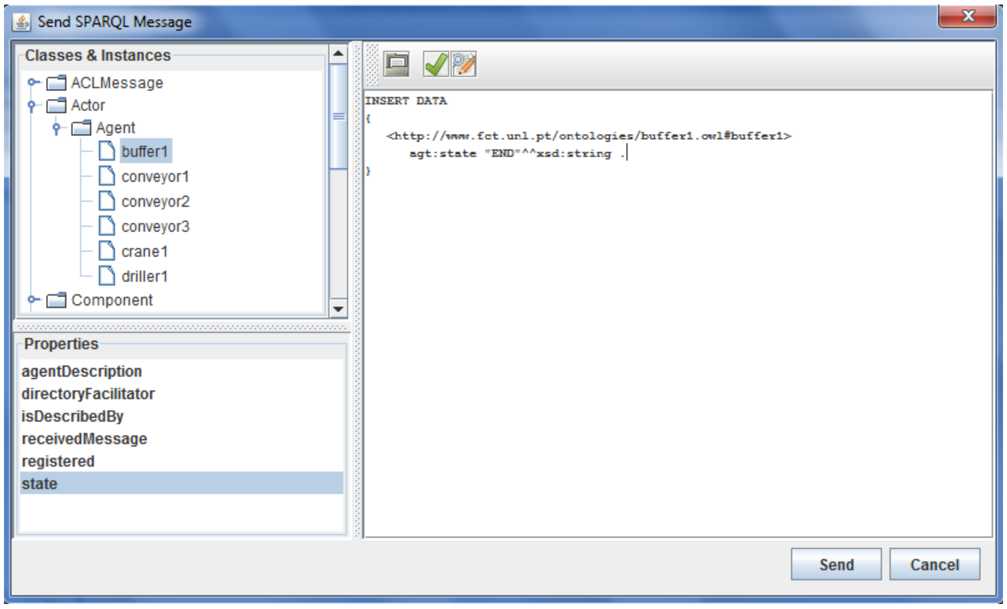
When loading an agent, a dialog to choose a file will appear. The user may then choose the *.properties* file that refers to the information needed to create the new Process Agent. This file contains contains four fields, three of which are used as the initial arguments to initiate the agent. These fields are:

- **agent.name** - The name that will be given to the newly created agent. This name will appear in the agent's AID.
- **agent.ontology.uri** - The URI of the ontology the agent needs to load. This field is used by the Process Agent to bind it with the loaded ontology.
- **agent.ontology.file** - The path to the file containing the ontology in OWL format. The Process Agent will open and load this file into its KB.
- **agent.ontology.main** - The URI of the instance of the *Agent* class that ontologically represents the Process Agent to load. As previously stated, this is used by the agent to gain an identity of its own.

When the file is read, the Monitor Agent gets the above mentioned data and uses it to initialize the Process Agent by filling its initial arguments, as described in the previous section.

To unload or kill an agent, the user simply has to click on the corresponding menu item and the action will be performed on the selected agent. Unloading the agent performs a *clean* finalization on it, while killing agent will simply take it down, as described in the previous section.

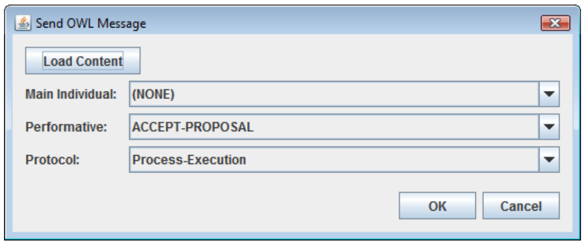
The menu bar in the GUI also has an *Actions* menu, shown in Figure 6.14b, which provides two more options that allow the user to send messages to the selected Process Agent. The *Send SPARQL/Update Message* option allows the sending of a SPARQL/Update message to the selected agent in order to update its KB. When this item is clicked, a dialog is opened, containing the SPARQL/Update editor shown in Figure 6.15. This editor provides an easy-to-use interface to edit and search SPARQL terms. When the user has finished editing the message and clicks the *Send* button, the update is sent to the selected agent and performed in its KB.



```
INSERT DATA
{
  <http://www.fct.unl.pt/ontologies/buffer1.owl#buffer1>
    agt:state "END"^^xsd:string .
}
```

Figure 6.15: The SPARQL editor in the Monitor Agent. This editor provides a tree to search for classes and instances (upper left), a list to search for properties (lower left) and a text area to edit the update. Clicking twice on the classes, instances or properties will make them appear in the text area. It also provides validation of the syntax used in the update, as well as the option to load it from an external file.

The *Actions* menu also has the *Send OWL Message* item, which allows the user to send a pure OWL message to the selected Process Agent. When this item is clicked, the dialog in Figure 6.16 is shown, where the user is prompted to select a pre-created file containing the ontological description of the message contents, select the message performative and the message protocol. When the *Send* button is clicked, the message is sent to the selected Process Agent.



Main Individual: (NONE)
Performative: ACCEPT-PROPOSAL
Protocol: Process-Execution

Figure 6.16: The OWL message chooser in the Monitor Agent. This dialog allows the user to load the contents of the message and choose its main individual, performative and protocol.

The described agent proved very useful for managing and monitoring Process Agents. Since it contains the complete descriptions of each ontology in the system, debugging and error control was much easier during the implementation of the proposed system. It also quickened the process of loading and shutting down agents in the system.

6.3 The Process Engine

The work for this thesis was mostly implemented by means of processes. Chapter 5 described the theory behind the current process implementation and the resulting OWL class ontology of this theory was detailed in Section 6.1.2. In this section the actual implementation is described and how it was integrated with JADE and Jena.

The engine is responsible for loading, managing and executing processes inside a Process Agent. It loads the process definitions from an ontology file and converts them to executable objects within the agent. In Figure 6.17 a detailed depiction on how the engine works is shown.

When the ontology is loaded from the file chosen for the current Process Agent, the engine reads all the descriptions of the processes associated to the actor represented by the agent. All those descriptions are converted to Java description instances and stored in the

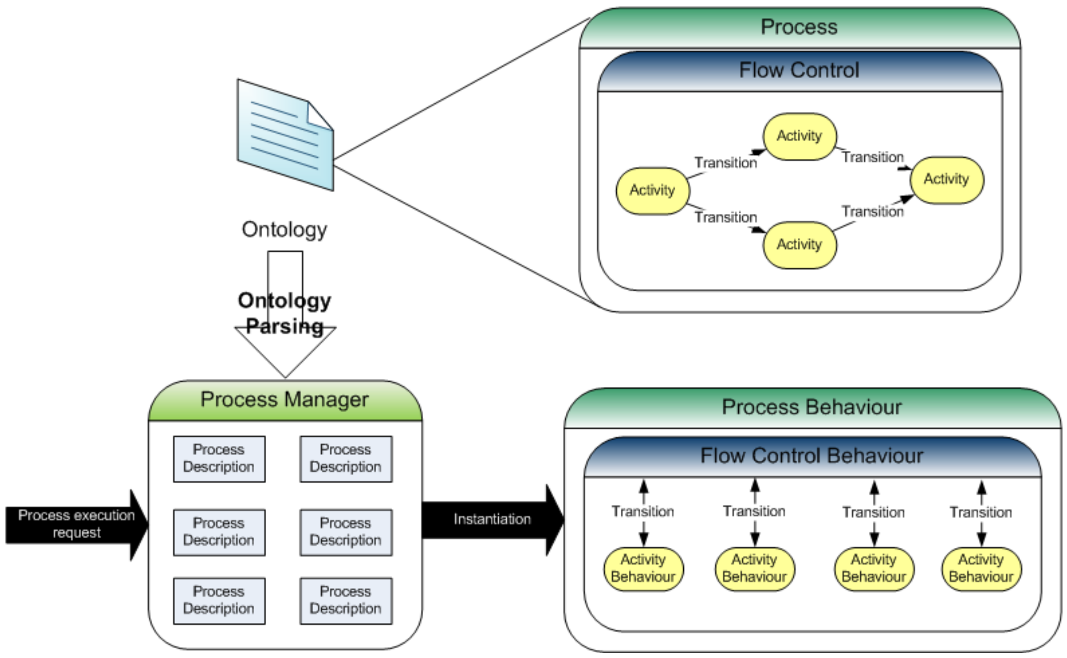


Figure 6.17: The conversion of a process described in OWL to JADE behaviours. This image shows how a file containing an OWL description of processes is parsed into the process manager, in the engine, and then instantiated to behaviours at run-time.

process manager instance of the agent. This conversion to instances in Java was devised for performance issues: When a request comes to execute a certain process, its faster to read its description from an instance in Java than to read it directly from the KB. The manager will then hold all the descriptions of the processes, which, in turn, contain the descriptions of the nodes associated to them, mirroring what was already described in the loaded OWL ontology file.

After parsing the ontology and having all the processes in memory, the manager is then able to receive requests for execution of its contained processes. If such a request comes, the manager checks if it contains the required process and if the correct inputs are filled. If that is the case, it instantiates the corresponding JADE behaviour, which will instantiate the needed node behaviours during execution, as needed.

In Section 3.1.4, it was stated that JADE models all of its tasks as *behaviours*. These were ideal for the implementation of processes, since their modelling is node-like, as is the process model in this work. Therefore, each process, flow control or activity is represented as a JADE behaviour.

One may have noticed that the transitions changed from the process ontological description to the process behaviour. This is because all the transitions are checked directly by the flow control behaviour containing them, that is, during execution, whenever a node has finished its own execution, the parent flow control will make a check for the transition from it to the next node.

Processes have a temporary memory which is only active in run-time. After the execution of a given process, there is no use to have it occupying space in the KB of the agents. Therefore, for easy removal, each process has its own KB, which imports the KB of the base agent. Activities need also private KBs as they may need to store temporary values in it, like their inputs, which are only valid during their execution. The private KB of an activity imports the KB of the corresponding process. In Figure 6.18 a KB tree can be seen, explaining all the imports.

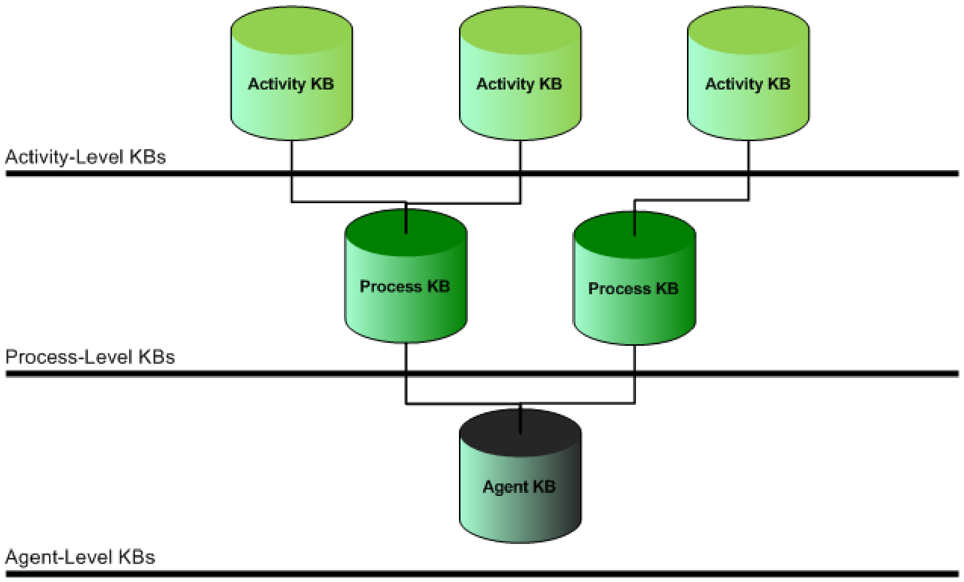


Figure 6.18: The KB tree inside a process engine. Below there is the agent KB. The processes import this KB. On the top level the activities contained in a given process import its KB.

As can be seen in the tree in the figure, the processes have access to the values in the agent's ontology, while the activities have access to the process' and agent's ontologies. This way, when a process behaviour finishes its execution, its KB is removed from the tree along with the KBs of its activities. For logging purposes, the contents of that KB are stored in the corresponding execution log for a user to be able to debug the agent.

No activities were implemented directly in the engine. As was stated both in Chapter 5 and Section 6.1.2, the defined models for activities are only templates that should be sub-classed for implementations. In the current system all the activities used were implemented in external libraries and described in external ontological files. The framework provides abstract Java classes that should be extended to implement activities: *RActivityOneShot*, *RActivityRequestInitiator* and *RActivitySimple*. These classes were created to mimic the *OneShotBehaviour*, *AchieveREInitiator* and *SimpleBehaviour*, respectively, of JADE [Bellifemine et al., 2010], while maintainig compatibility with the process framework. An example of an activity implementation is shown in Listing 6.5.

Listing 6.5: Template to implement an activity. This template may be used to implement a *RActivityOneShot*. All the code within the activity should be placed inside the *action* method.

```java
public class MyActivity extends RActivityOneShot
{

    public MyActivity(ProcessAgent agent, RProcessBehaviour process
                                          , OActivity oActivity)
    {
        super(agent, process, oActivity);
    }

    @Override
    public void action()
    {
        //Activity implementation.
    }
}
```

When all the activities are implemented, by creating subclasses of the provided templates and filling the needed methods, and their respective libraries are generated, the *config/config.properties* file in the agent's root directory, needs to be edited to point to the location of the newly created *.jar* files.

Designers have also to create the ontological descriptions of the new activities. To do this, they need to import the process ontology, create a subclass of the *Activity* class and provide the necessary properties, as seen in Figure 6.19. The created ontology with all the activities has, then, to be imported by the ontology the designer wished to create the activity instance in. For example, if the activity is used in a given ontology of a manufacturing module, it needs to be imported into that ontology.

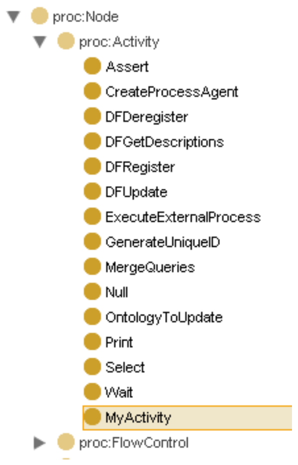


Figure 6.19: A new activity as a subclass of the *Activity* class. This image was taken from the Protégé editor.

When all of the above steps are complete, the next Process Agent that is instantiated will dynamically load the custom activities and be instantaneously able to execute them without any kind of reprogramming. A set of default activities was created for the current implementation which will be described in Appendix A.

For a better understanding of process configuration and execution, in Appendix C the creation of a simple example process is explained.

6.4 The Process Editor

Even though the processes in this work have a well-defined model for implementation, along with many features to simplify their creation, using a normal OWL editor like Protégé would render process creation very complex. Process creation generates rather large ontologies that might easily confuse a designer. For this reason, a GUI was created to allow the designer to create processes for agents using the previously described specification. This editor manages processes in a single ontology and is similar to most of the existing software in the market, so its use is easy to master.

The designer may create new process ontologies or alter existing ones. The processes are shown in a tree on the top left corner of the interface. In the bottom-left corner, the available activities are shown and the user may drag them to the selected process in the center of the frame. The main window of this editor can be seen in Figure 6.20.

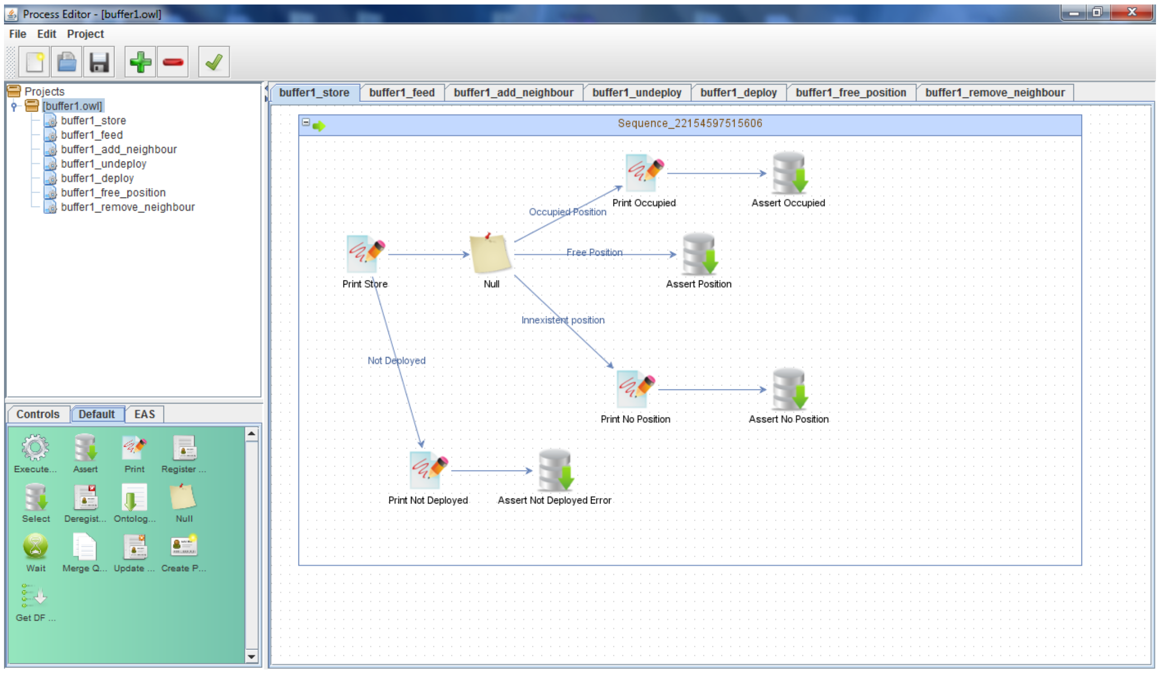


Figure 6.20: The main window of the Process Editor.

This editor also supports viewing the textual representation of the ontologies and each process, activity, flow control or transition can be configured via dialogs provided by it. A SPARQL editor, identical to the one shown in Section 6.2.2, was also implemented in this tool to ease the creation of queries.

Even though this editor was only designed to create processes, the OWL file generated by it may be later edited using external tools, like Protégé, to complete ontological definitions. There is no problem whatsoever of loading the edited file again by this program. It is even recommended to do so, in order to add more terms to use in the SPARQL editor.

Just like in the process engine, new activities can be dynamically added by loading a *.jar* and an ontology file with their description. The process of creating new libraries for the editor slightly different, since the new library will not have the actual implementation of the activities. Instead it will contain the implementation of the configuration panel of the activity to use in the editor. It will do basically the same as the process engine to load external activities. It also has a configuration file in its root directory in which the user may insert the path to the *.jar* file of the library. An example of an activity implementation is shown in Listing 6.6.

Listing 6.6: Template to implement an activity in the editor. The method *delete* is invoked when the activity is removed from the process and where the code to remove the correct properties from the ontology should be. The method *getConfigurationDialog* must return a dialog programmed to correctly configure the activity.

```
public class MyActivity extends Activity
{

    public MyActivity(ProcessEditor editor, NProcess process
                                    , Individual node)
    {
        super(editor, process, node);
    }

    @Override
    public void delete()
    {
        //When an activity is deleted from a process,
        //this method is called.
        //A designer may override it for a more complex deletion.
    }

    @Override
    public ComponentDialog getConfigurationDialog()
    {
        //This method should return a user-created configuration
        //dialog for the activity.
    }
}
```

For further explanation on this subject, Appendix C, as well as exemplifying a process creation, also demonstrates some of the features of the Process Editor, like the SPARQL editor dialog.

6.5 EPS System Implementation

The described MAS was used to implement a simple control structure based in process execution in the scope of EPS, which was already described in Chapter 4. This system was implemented using the MOFA France kit components serving as a basis for all the manufacturing modules. It is generally used for testing implementations of projects in the manufacturing area. An image of this kit is can be seen in Figure 6.21, showing conveyor belts, drillers, a crane and a buffer, along with other modules.

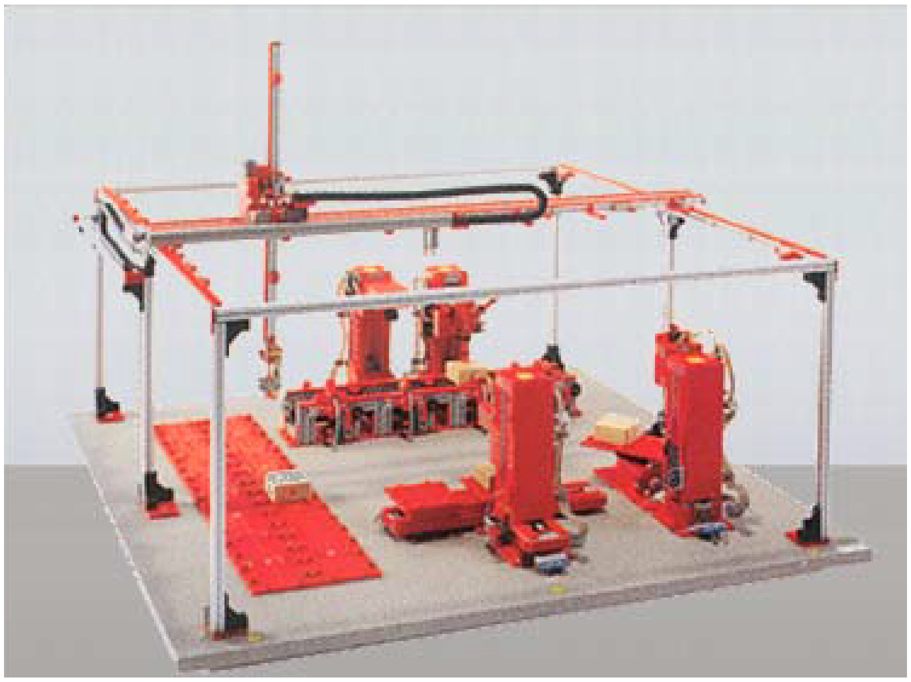
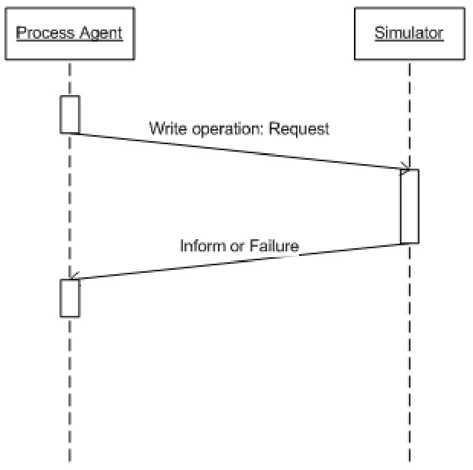


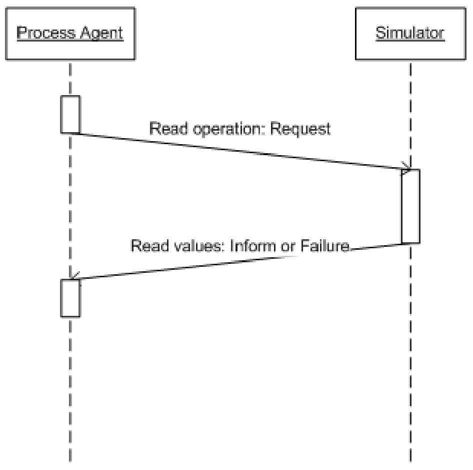
Figure 6.21: The MOFA France kit, deployed in its default configuration for simulation purposes. This kit provides several test components, like conveyor belts, cranes, machine tools, buffers or machine tables.

Rather than using the actual kit, a simulator based on its components was created. The reason for this was to simulate module repositioning to achieve evolvability in the system. Although the current implementation was created to work with the simulator, it can easily be reconfigured to become compatible with the physical kit: one simply has to edit the activities responsible for interfacing with the simulator to interface with the actual kit. The mentioned simulator is described in Section 6.5.5.

To emulate driver reading and writing, two custom activities for this purpose were created specifically for this system. These activities, instead of sending messages to the actual drivers of the kit, are programmed to send ACL messages to the simulator using the FIPA Request Interaction protocol. Contrary to all the other messages in the system, these do not use ontologies. They use simple commands in String format for simplicity purposes. The most commonly used requests are to *write to* the drivers, which send orders to a given component in the simulator, and *read from* the drivers, which query a component for information about sensors and positions. Both of these interactions are shown in the UML sequence diagrams in Figure 6.22.



(a) The write request.



(b) The read request.

Figure 6.22: The requests to exchange information with the simulator. Messages are exchanged using the FIPA Request Interaction Protocol.

Another topic to consider is, even though the system supports user-defined message exchange, all the interactions between the modules were implemented only by external process invocation. In other words, whenever a module needs to send information in order for another module to process it, it invokes the corresponding process, which was already shown in Section 4.2.

Several generic processes were implemented which are common to most of the modules. Even though their implementation may vary from module to module, the objective is the same. These processes are shown in Table 6.1.

Table 6.1: Common processes to all the modules.

Process	**Description**
Add neighbour	Adds a neighbour module to the list of neighbours. Neighbour modules may perform skills together.
Remove neighbour	Removes a neighbour module from the list of neighbours. All interactions with this neighbour will cease.
Deploy module	Deploys the module in a given coordinate of the shop floor, allowing it to find new neighbours.
Undeploy module	Undeploys the module from its current position in the shop floor, removing all its previous neighbours.

Neighbour-related processes are only implemented by material handling modules. In the current implementation, modules that may provide product transporting or storing, need to be aware of neighbours with whom they may interact in order to handle the products. Neighbour search occurs upon deployment, when modules become aware of their position in the shop floor. When a module is deployed, it searches for modules whose positions are compatible with the execution of transporting, storing or feeding skills. If that is the case the module invokes the *Add Neighbour* of the new neighbour and that fact is asserted in each others KB. A generic deployment process involving two modules is shown in Figure 6.23. Note that in some modules this process might be a little different.

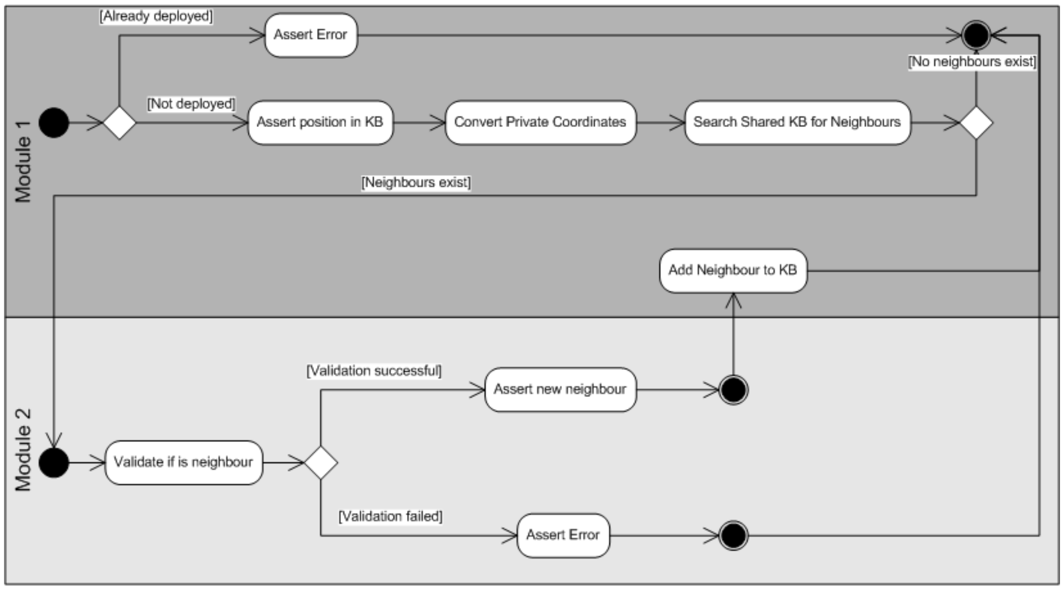


Figure 6.23: A generic deployment process. The deployed module searches for neighbours and, when one is found, it invokes the other module's *Add Neighbour* process.

In the same figure, the process in which the other module adds the current module as a neighbour when it is requested, can also be seen. When both the processes are finished, all the modules should have asserted the new neighbours in their KBs and are then ready to perform skills together.

The undeployment of a module is the exact opposite of a deployment process: It removes the module positioning in the shop floor and removes all the neighbours from its KB, invoking their *Remove Neighbour* process so they may remove the module from their own KB.

All the modules described in the following sections have an ontological description with the properties of the components they represent as well as the processes they need to load. This description is loaded to a Process Agent that will be in charge of controlling the module.

In order to accommodate new concepts related to manufacturing, a new ontology was created for this particular system. More details on this ontology are provided in Appendix B.

6.5.1 Conveyor Implementation

The conveyor Process Agent implements the functionalities of a given conveyor belt in the system. The MOFA France conveyors are equipped with a sensor, which is activated whenever there is an object on them, and are able transfer products between other conveyors of the same type. The skills considered for this agent are described in Table 6.2.

Table 6.2: The skills of the conveyor agent.

Skill	**Description**
Transport	Transports a product from the conveyor to a neighbour conveyor or receives a product from the neighbour conveyor.
GetPosition	Gets an available position to insert a product in the conveyor. A conveyor only has one position where products may be placed.
Feed	Causes the module to release its handling off a given product and sets the conveyor position as free.
Store	Stores a given product in the conveyor, marking its position as occupied.

Since conveyors are able of influencing the position of a given product, they are considered material handlers. In the implemented system, the they may interact with the crane and other conveyors to transport a product. This is achieved by conveyors adding the nearby modules as neighbours.

A skill that is worthwhile describing is the one that allows conveyors to transport products between themselves. It requires two conveyors to be executed: a source conveyor and a target conveyor. Both of the conveyors need to be neighbours. An UML sequence diagram representing this skill is shown in Figure 6.24.

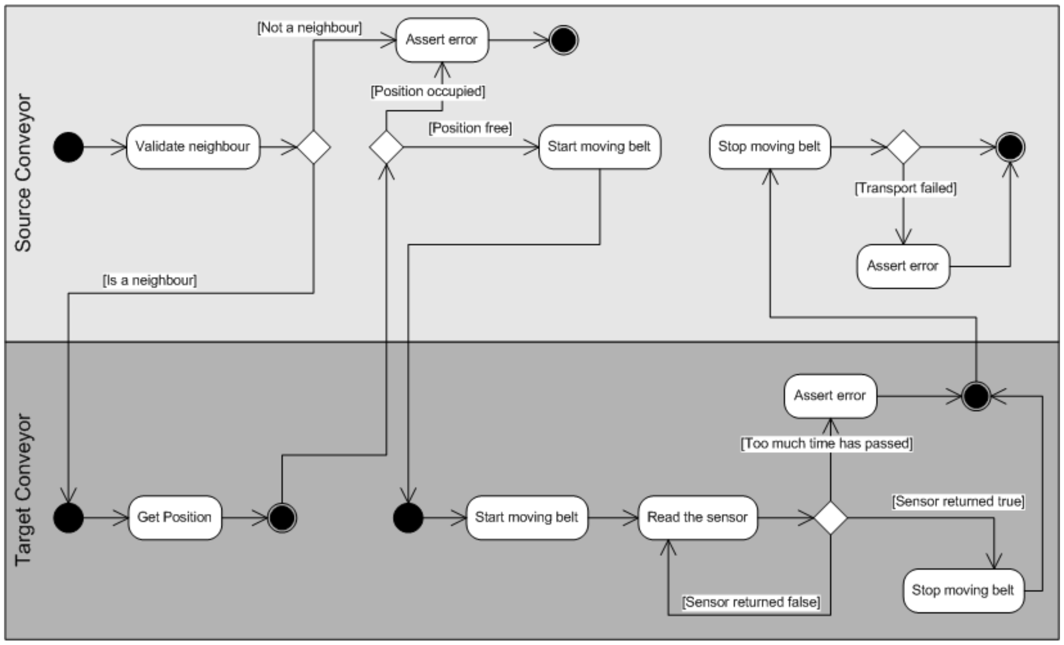


Figure 6.24: A diagram of the transportation skill in conveyors. This UML sequence diagram shows how two conveyors transport a product between each other.

The transportation diagram shows how the skill works both for the source conveyor as well as for the target conveyor. It involves also the skill used to get a position from the conveyor in order to check its availability. If any of the processes fail, an error is asserted into the KB of the agent.

6.5.2 Crane Implementation

This agent allows the control of a MOFA France crane. The crane may pick a product from anywhere in the system, as long as it is able to reach it, and place it in another location. The skill provided by the crane agent is shown in Table 6.3.

Table 6.3: The skills of the crane agent.

Skill	**Description**
Pick and place	A sub-set of the transport skill. The crane uses this skill when it comes for it to transport a product from one location to another.

A crane is a material handling module capable of transporting a product but may not store it in its position, therefore the main skill of this agent is the pick and place skill. It requires synchronization with the source target modules. An activity diagram depicting the functioning of the skill can be seen in Figure 6.25.

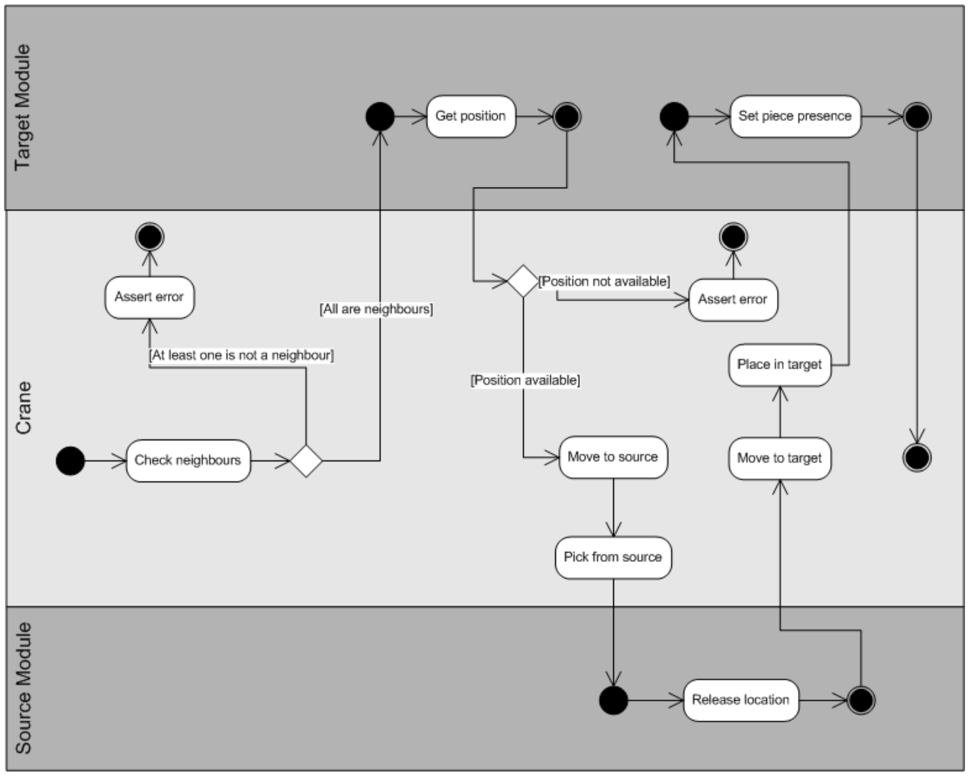


Figure 6.25: A diagram of the transportation skill in cranes. The crane interacts with both the source and target modules.

The source and target modules in the figure were kept generic in their functioning, as their execution of the assigned tasks varies from module to module. It also shows that the crane will invoke the *Feed*, *Hold* and *GetPosition* of other modules in order to transport a product. If an error occurs during the execution, it is asserted in the KB for later recovery, if needed.

As for neighbours, the crane will accept any module that possesses the needed skills for a pick and place, whose position is contained inside its range of movement. It is worth to note that conveyors next to a driller will not be able to become neighbours of the crane, since the path to their position is blocked.

6.5.3 Buffer Implementation

A buffer may store products in one of its available positions or feed products to the system from one of the same positions. In the current system, the products processed inside a cell start off inside the buffer and also end inside it. The skills supported by the buffer agent are shown in Table 6.4.

Table 6.4: The skills of the buffer agent.

Skill	**Description**
Feed	Outputs a piece to the system.
Store	Inputs a piece from the system.
GetPosition	Gets an available position from the buffer.

Even though this module is not able of actively transporting products, it handles products by keeping them in one of its positions. This way it influences such transport operations with its *Feed* and *Store* skills, making it also a material handling unit.

Since the buffer has more than one position in which the products may be placed, all of the skills inside this module have a more complex behaviour than the ones in the conveyor, having to choose from one of the positions.

In the current configuration, the buffer is only able to add the crane as its neighbour for skill cooperation, since it is the only module that can take a product out or put a product in one of its positions.

6.5.4 Driller Implementation

The driller agent controls one of the machine tools in the MOFA France kit. The driller is the only machine that actually performs some sort of transformation in the products. The skill performed by this module is shown in Table 6.5.

Table 6.5: The skills of the driller agent.

Skill	**Description**
Drill	Drills a hole in a given product.

The drilling skill of this agent is a simple skill where the machine simply moves towards the product and rotates the driller. Being this a skill that influences the physical properties of a product, this module is considered a transforming unit and not a material handling unit, like the previously described modules.

Though it is not able to transport a product anywhere, all the transporters in the system will work to transport a product towards it.

6.5.5 EPS Monitor Agent

In order to facilitate the deployment of the agents in the EPS system, the EPS Monitor Agent was designed. It allows the designer to visually place all the other Modules in the system in a physical place within the shop floor. This eases the testing of different layouts. The main interface of this agent can be seen in Figure 6.26.

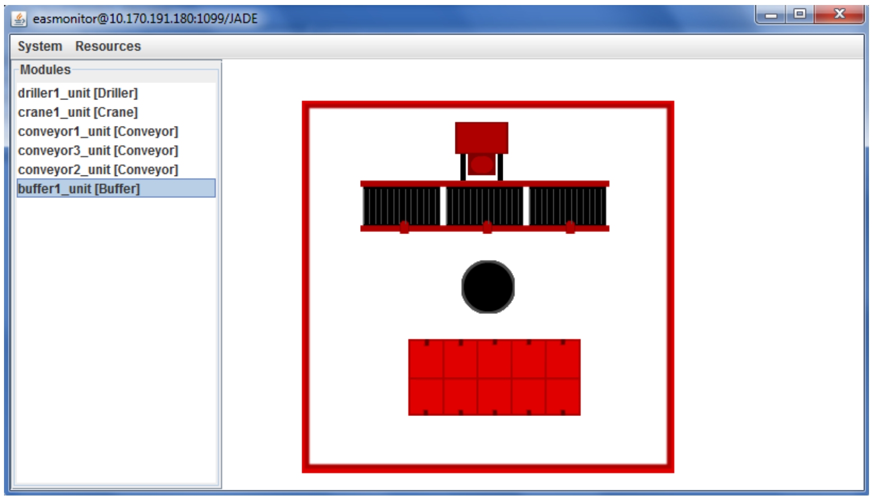

easmonitor@10.170.191.180:1099/JADE
System
Resources
Modules
driller1_unit [Driller]
crane1_unit [Crane]
conveyor1_unit [Conveyor]
conveyor3_unit [Conveyor]
conveyor2_unit [Conveyor]
buffer1_unit [Buffer]

Figure 6.26: The EPS Monitor Agent GUI. It allows the user to drag modules into the virtual shop floor and to create products for simulation purposes. The modules show animations in accordance to the orders received.

This agent also behaves as a simulator for the deployed modules. When the modules activate their drivers in order to execute certain skills, the GUI of this agent shows the animations. The designer can test what happens to products fed to the system by adding one to any of the material handling units. This visual representation of the system eased the implementation of the other agents.

As mentioned earlier, the implemented system does not communicate with the physical kit, because, with this agent, the user is able to put the modules anywhere and with different rotations, which simulates the adding or removal of physical components of the system as well as repositioning.

The need of an agent like this became clear when working together with the student Nuno Pereira to create an EPS. Since the the author's colleague was in charge of the algorithm for repositioning and control of manufacturing cells and his work was independent from the one in this thesis, a need arose to simulate the placement of modules, which was incomplete at the time, as was also this implementation.

6.6 Testing Scenario

To validate the implemented system two testing scenarios were created by means of two cells with different behaviours.

In the first scenario, a cell agent was configured with a single process that allowed products to start in the buffer and be transported to a driller, where they were processed. These products were then transported back to the buffer to be stored. Figure 6.27 shows the used configuration.

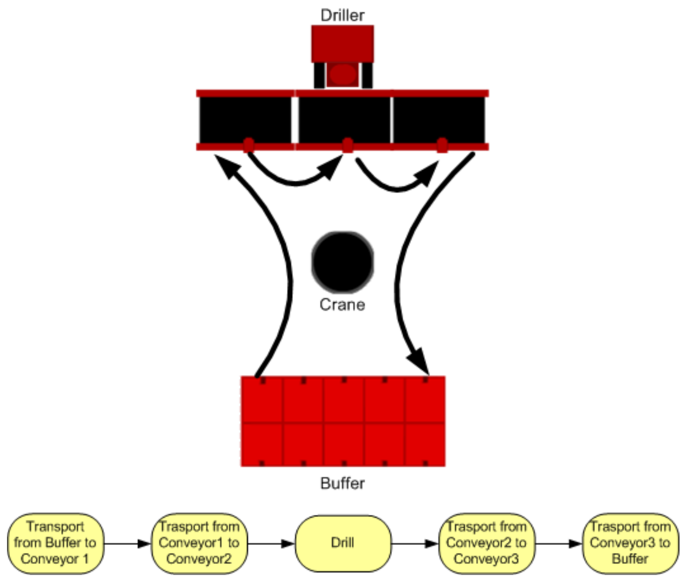

Driller
Crane
Buffer
Transport from Buffer to Conveyor 1
Trasport from Conveyor1 to Conveyor2
Drill
Trasport from Conveyor2 to Conveyor3
Trasport from Conveyor3 to Buffer

Figure 6.27: The first testing scenario with the initial shop floor configuration. It also shows the production process below.

The created cell agent contained a series of references to the products and to the modules contained in it. Whenever products were in their initial state, the cell invoked the transportation skills of the correct modules to carry the products to the conveyor in front of the driller. When the products were in place, the driller would be invoked by the cell to perform its drilling skill. After the products were stored back in the buffer, the cell would set their state to the final value.

This is the most static of both scenarios, although it allows a great degree of reconfigurability. When a module is added or removed from the system, or a different path has to be chosen, the designer only has to correct the process of the cell to cope with this change. In Figure 6.28 can be seen that, when the initial configuration was changed, the corresponding process had to be adapted. Those adaptations are shown in red.

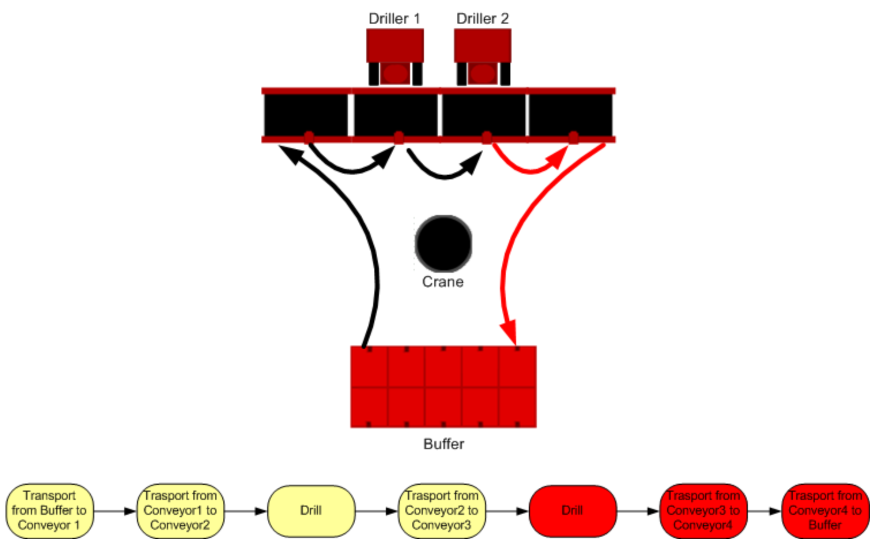

Driller 1
Driller 2
Crane
Buffer
Transport from Buffer to Conveyor 1
Trasport from Conveyor1 to Conveyor2
Drill
Trasport from Conveyor2 to Conveyor3
Drill
Trasport from Conveyor3 to Conveyor4
Trasport from Conveyor4 to Buffer

Figure 6.28: The first testing scenario with the shop floor configuration after a change. It also includes the change in the production process below.

In the second testing scenario, the cell relies of a plan to perform the correct operations on a product. For more information on the plans see Appendix B.

The plan contains the operations to be performed to the products, as well as information about the path towards the place were such operations are to be performed. In this scenario, any alteration to the plan does not require reconfiguration of the processes executed by the cell so they may be executed at run-time. In Figure 6.29 the basic set-up of the system is shown for this scenario.

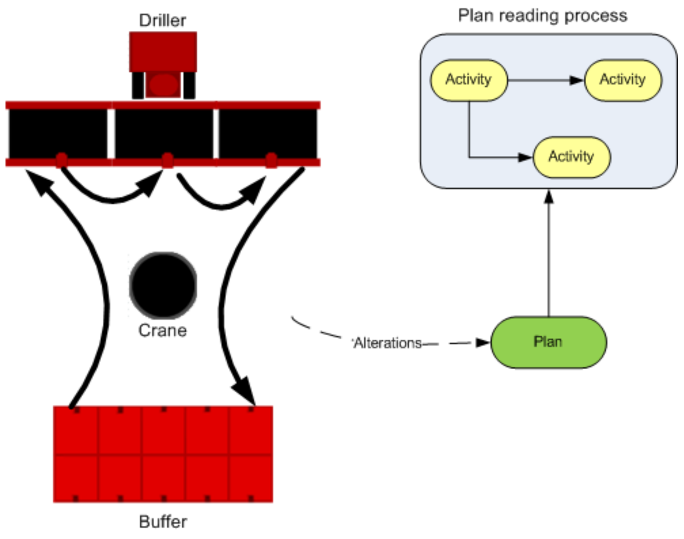


Figure 6.29: The second testing scenario, where cell execution is performed with the aid of a plan.

Although this approach relies on more generic processes and, therefore, less freedom to cope with certain specific issues of the system, it opens the way for dynamic reconfiguration of the production process by means of algorithms to calculate paths whenever a disturbance in the system occurs.

An important note that is worth referring is that, when creating both test scenarios, no hard-coding was needed. It was only required to create new ontological configurations for the different cells.

These actions were all accomplished using processes in all the different agents in the system. Even though these were simple testing scenarios, the theory and implementations backing them up were complex. The author considers that, as simple as this might be, his objectives were achieved, by proving that an EPS could indeed be controlled using the concept of BPM.

Chapter 7

Conclusions and Future Work

During the previous chapters, it was proposed a new way of controlling manufacturing components in a shop floor, resorting to a process-based MAS with support for ontological descriptions and following the EPS paradigm. This culminated in an implementation of a framework that was validated by two test scenarios.

This chapter concludes this document, summarizing the work done in this thesis and discussing the results achieved. It also proposes future work in this area to cope with more problems and extend the capabilities of the technologies here developed.

7.1 Conclusion

Recent works in the manufacturing area tend to rely more on distributed architectures and ontological descriptions to control production and assembly systems. However, companies also have different areas other than the shop floor, like marketing or product planning which share applications so that they may become more agile by use of BPM.

In recent years, companies, whether large or small, have relied even more in BPM to achieve inter-operability and easy reconfiguration of their processes, resulting in a growth of demand for more and better BPMSs. Nevertheless, there are many areas within companies still mostly ignored when applying this concept, whether its because current process-based systems still only support higher-level activities or because actual implementations may require more than simple adaptations. This is the case of the manufacturing shop floor, where control architectures tend to be implemented using more traditional ways.

Current research that applies BPM to a manufacturing context does so at a very high level within the company, leaving out the machine control during the production process. In this context, this work focused on a new approach that applies process execution and analysis to the shop floor, as a supporting technology for an EPS. This merging of two seemingly unrelated concepts was ideal to achieve easy reconfiguration, which is highly stressed when discussing distributed manufacturing systems.

The idea of applying process execution within the shop floor environment became apparent at a very early stage of this work when a way was needed to create skills within manufacturing modules that might be easily reconfigured in order to keep the system generic. A first approach was do define them as rules, using a rule engine, but this was discarded due to the fact that it would require a great amount of rules to create a single skill, since they are merely composed of *if-then-else* statements and have limited syntax for more complex behaviours. Another approach was to use a separate scripting language to define them but that would add complexity to skill creation when the objective was the exact opposite. Finally, and after some research, the process-based approach was chosen, due to the fact that it can generate complex behaviour in a simple way and is easily configurable.

Initially, the only intent was to generate the skills in modules using the BPM paradigm, while keeping other interactions, like the additions of neighbours, hard-coded. But, as the time passed and new developments were made, it was clear that it was viable to spend an additional effort in creating a generic system solely based on process execution and create all the interactions in the manufacturing context on top of that, by means of configuration instead of coding.

The use of a MAS and ontologies as the lower level technologies of this system was a decision that stood from the beginning, with the help of Professor José Barata and Dr. Regina Frei. This would also allow the system to remain generic, being configured through ontological descriptions, and cope with changes in a modular way, by only affecting the correct agent. So, the objective was to create a BPM-based system on top of both technologies to make the best of their advantages.

The EPS to control the manufacturing shop floor components was then created on top of the process-based technology developed in this thesis, by using the advantages of easy configuration. This system was very simple, when compared with actual implementations in previous researches. This was because a new approach was being validated, which could be further developed in future works. The main focus in this thesis was having a manufacturing system working with a process-based MAS, and therefore, a great deal of effort was applied in creating a process execution framework compatible with OWL and a Multi-Agent platform, in this case, JADE.

During the implementation and validation of the proposed system, the modularity provided by the use of a MAS proved priceless in terms of task assignment to specific entities and a more localized configuration. When it came to adjust the system to work in a certain way, most of the times, there was only need of configuring a single agent. Also, the fact that agents possess well-defined communication protocols and languages provided a solid basis for interactions within this system.

The use of ontological descriptions in OWL to represent a manufacturing system, along with SPARQL to query KBs provides extensibility to the work of the designers. By relying on ontological descriptions to map the current states of the system, instead of hard-coding them, it was possible to easily handle complex changes in KBs. Therefore, equipment may be dynamically added or removed to the system, processes may be changed or agent behaviour may be modified by only affecting the ontological descriptions.

The BPM-based approach proved to ease the configuration of each manufacturing module, by providing concepts for easy-to-use creation and diagnosis tools. Reconfigurability reached a level where an agent could attain a completely different behaviour by resorting only to the edition of a single ontological file containing its process definitions.

The test cases to the solution presented in this work proved that it was agile enough to cope with changes in descriptions and manufacturing processes without the need to completely reprogramming the agents. Therefore, the agility objectives previously proposed in the beginning this thesis (Modularity, programming effort reduction, re-usability and self-organization) were achieved. Another result worth mentioning is that, the fact that this was a BPM-based approach, made it possible for the developed process model to be used in areas other than manufacturing, allowing a company-wide communication between departments to follow the whole production process.

Several months of research led up to this work, analysing and testing many different approaches and technologies, some with more success than others. During this time, two documents were co-written by the author detailing the research accomplished up to that point. A technological report was created for the Birkbeck College in London and an architecture was presented in the Industrial Electronics (ISIE), 2010 IEEE International Symposium containing earlier versions of this work [Frei et al., 2009, Frei et al., 2010].

7.2 Future Work

In this work, the system proposed is a simplified version of a system with BPM capabilities. As stated before, there are other more powerful BPMSs in the market. As simple as it is, this system provided a way to merge BPM with ontological descriptions in OWL and MASs in JADE.

Further work with the core process engine could include extension to provide full diagnosis for process execution. Processes could be debugged at execution-time so the designer could see in real time the inputs and outputs of each activity. Also, benchmarks and statistics could be gathered at run-time.

The implemented process engine is still very much data-driven: The designer has to know a priori all the concepts related with its functioning in order to work with it. The author proposes a new version, which is more event-driven, abstracting the data and providing support for asynchronous messaging. Another addition to the engine could be the use of policies to help boost process execution and creating safety rules within the system without affecting the processes themselves.

The presented system already supports OWL and SPARQL and, to aid with these technologies it is used the basic reasoner Jena provides. This reasoner is a relatively simple one and with many limitations but it was the only option due to incompatibilities between SPARQL/Update and other more powerful reasoners, like Pellet. These problems will eventually be fixed. The use of better reasoners would provide extra extensibility to this system, since they would allow faster reasoning and the use of extra OWL-related technologies, like Semantic Web Rule Language (SWRL), used to create rules in ontologies.

The implemented EPS was fairly simple because this work threaded in relatively new grounds for manufacturing inside the shop floor. The author proposes a future implementation using more complex ontological descriptions and processes in order to make the best of the concept of EPS. These descriptions could help coping with different problems in the manufacturing context, like diagnosis, error recovery or Product Lifecycle Management (PLM) [Sudarsan et al., 2005], just to state a few.

Appendix A

Custom Activities

In previous chapters it was stated that the developed process engine did not contain any default activities, although it could be provided with them by dynamically loading external libraries. So, in order to have a fully functional system, there was a need to create several base activities. This could prove both that this was a sufficiently agile system to control a manufacturing shop floor and that it was fully extensible, even in its atomic actions.

This appendix describes the two libraries containing activities created specifically for this work: The *Default* library and the *EPS* library. These libraries provided the base set of activities that helped create all the processes in the presented architectures.

The *Default* library consists of general activities that might be used in processes created for any area. This library includes generic activities such as printing or asserting facts in the KB. These activities are listed in Table A.1. All these activities have their implementation in a separated *.jar* file, which is imported by the system at run-time.

The other library was created to provide more activities directed to the context of the implemented EPS architecture, such as driver-related operations. This library contains activities that are more complex than the normal ones and involve more processing, like Cartesian calculations. The activities in this library are listed in Table A.2.

The possibility to create this type of libraries proves the extensibility of the presented system. If more activities were needed, they could be easily added via new ontological descriptions coupled with libraries containing the actual implementations. Also, as is the case in this work, several sets of activities may be created according to different domains for easier understanding and configuration.

Table A.1: The set of activities contained in the *Default* library.

Activity	**Description**
Assert	Uses a SPARQL/Update query to assert new knowledge in the KB.
Null	Does nothing. In many cases, a null activity is useful for a designer to keep the process readable.
Print	Receives a String as an input and prints it in the console.
ExecuteExternalProcess	Executes a process contained in a different actor, by using the default interaction protocol already discussed in Section 4.1.1.
SetLocal	Sets the value of a given local parameter.
SetOutput	Sets the output value of the process this activity is in.
GenerateError	Generates an error in the output of the process this activity is in. This is useful when the designer wants to create customized errors.
SendACLMessage	Sends an ACL message with several parameters and OWL content.
ReceiveACLMessage	Receives an ACL message, by comparing the input parameters as if they were templates.
GetSystemDate	Gets the current system date.

Table A.2: The set of activities contained in the *EPS* library.

Activity	**Description**
DriverRead	Sends a request to the simulator in order to receive data about a given module.
DriverWrite	Sends a request to the simulator containing orders for a given module to perform.
ConvertCoordinates	Gets the new coordinates of a given point, by translating it by a certain distance and/or rotating it by a certain angle.

Appendix B

EPS Ontology

Besides the ontologies already defined for the generic process-based system, there was need of configuring it to control a set of manufacturing components. To do so, they had to contain and understand manufacturing-related concepts. This way, an ontology was created that provided EPS-related concepts for modules to have a common language. The several classes and properties of this ontology is detailed in this appendix. One must understand that, if this explanation were to be done within the core chapters of this thesis, it would become rather bulky and long.

In Chapter 4, several concepts were described in order to create an agile and self-organizable manufacturing system, taking advantage of the EPS paradigm. The ontology here presented was describe these concepts. This ontology imports the already described process ontology to provide extra concepts and to not have to define them all over again. Here, relationships between manufacturing resources, skills and products are defined. The URI for this ontology is http://www.fct.unl.pt/ontologies/eas-system-ontology.owl.

This is the most complex description in the entire system, so, for readability purposes, the UML diagrams were divided into three categories (*equipment*, *modules* and *products and plans*), even though they belong to the same domain. In Figure B.1 is shown the section containing the description of the equipment.

Physical manufacturing modules have a set of static properties which describe their traits. The figure shows that the previously described modules (*Driller*, *Buffer*, *Conveyor* and *Crane*) have their own classes. All the descriptions of these modules are instances of the shown classes. As also can be seen, the *Equipment* class is a subclass of *Entity*, which

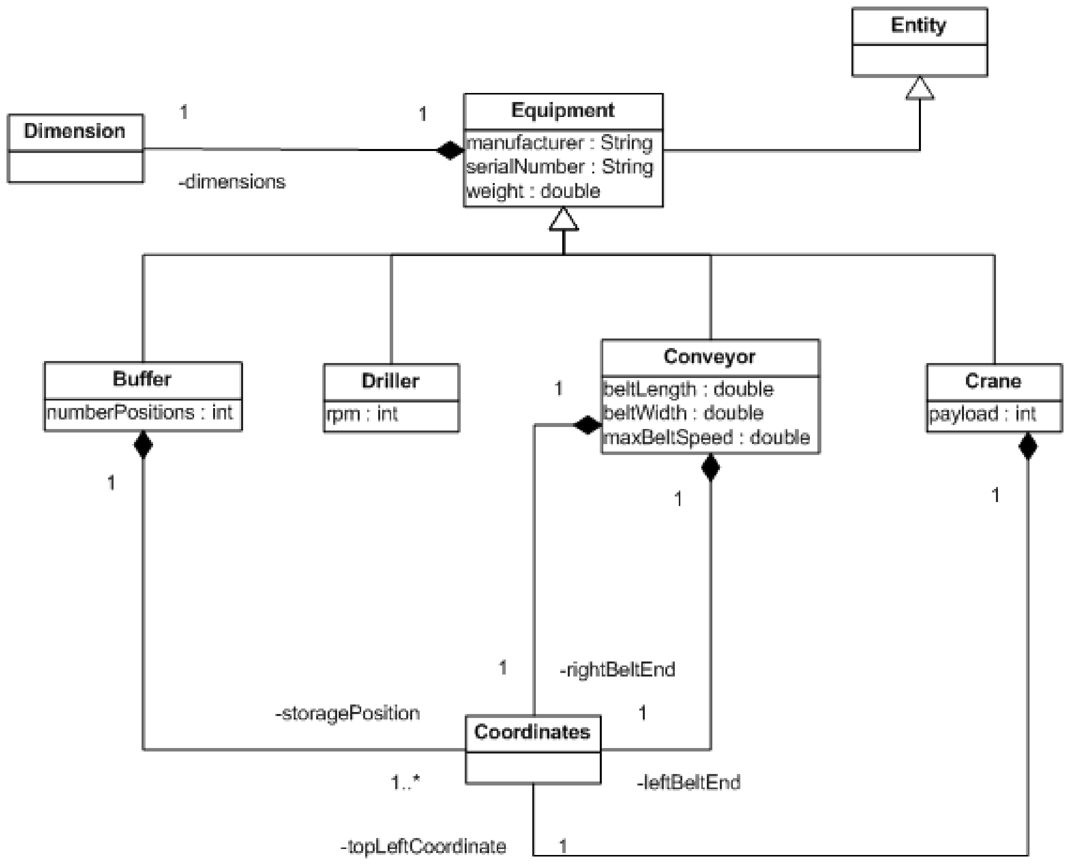


Figure B.1: UML class diagram of the equipment, showing all the classes and properties related solely to the physical equipment. Here, it can be seen the concepts that describe the equipment used to control the MOFA France components.

was already defined in the process ontology. Some of the classes also reference *Coordinate* and *Dimension*, both of these already defined in the concepts ontology.

As was described in Section 4.2, the system is composed of modules, which, in turn, contain skills that represent their abilities and add value to the manufacturing processes. To address this, the classes shown in Figure B.2 were created.

The diagram shows that both units and cells are subclasses of modules. One important topic in this ontology is the distinction between transforming units and material handling units. A transforming unit is a piece of equipment responsible for physically altering a given product, while a material handling unit aids in the transportation or general handling of a product in a shop floor. In this work the only transforming unit in the system is the driller, while the other components are material handling units.

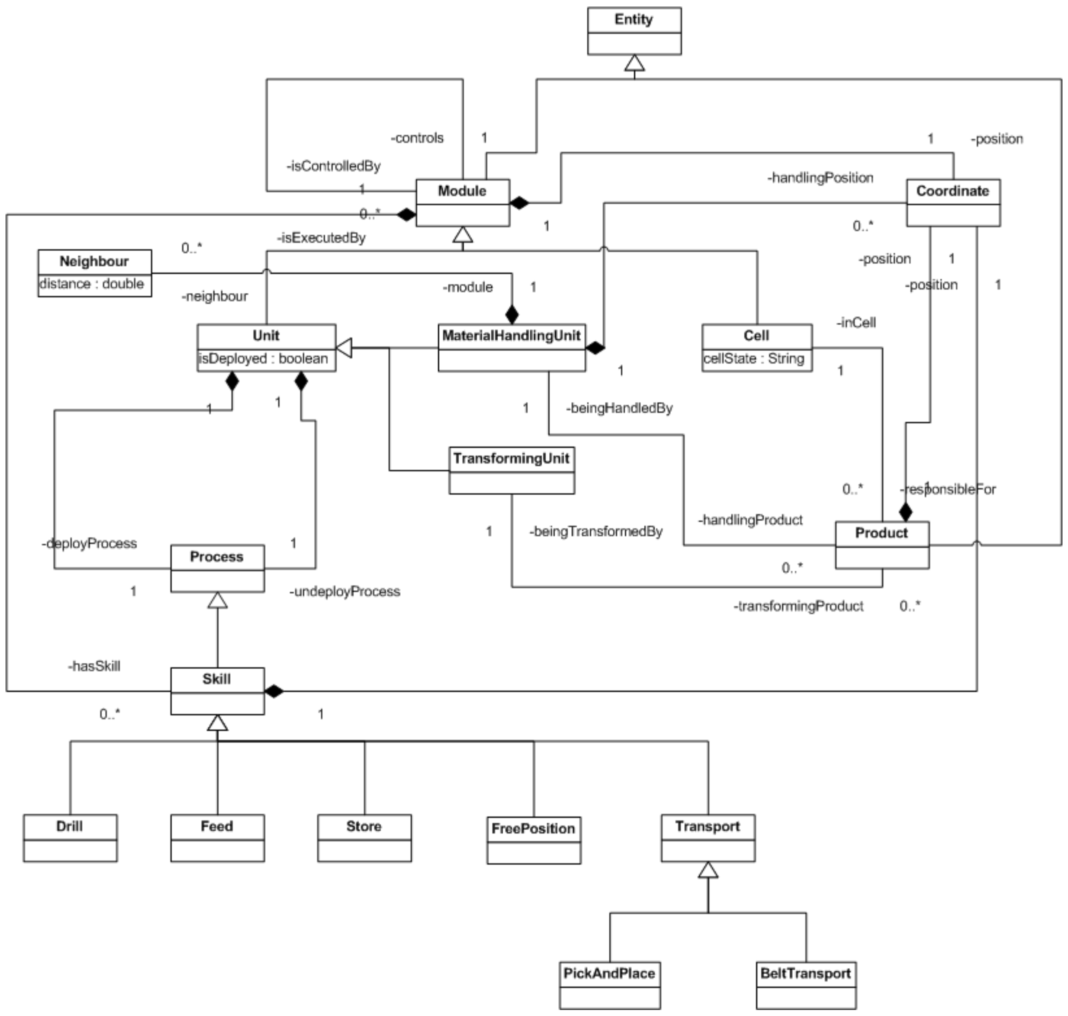


Figure B.2: UML class diagram of the modules and skills.

A material handling unit may have zero or more handling positions. These special positions are places where the products may be stored in that unit or key points for product transport. As an example, the buffer has ten handling positions. Also, material handling units may add other units as their neighbours, as well as their distance and relative position (whether the other module is above, below, on the left or on the right of the current module).

Cells are another type of modules, which may control and monitor a certain set of units and are responsible for a certain set of products. This type of module is responsible for executing a production plan and coordinating all the units it controls and is not directly associated with any physical equipment.

Independently of their type, modules may have a certain set of skills, which are a type of processes. Being a subclass of *Process*, skills may be directly invoked by the system and perform the tasks associated to them. In the current implementation, the defined types of skill were *Drill*, *Feed*, *Store*, *FreePosition* and *Transport*. Material handling units that may transport a product, have skills of the *Transport* class. If these material handling units can keep a product in one of their handling positions, they have skills of the *Feed*, *Store* and *FreePosition* types.

Another important part of this ontology is how cells manage their manufacturing plans for the products they are responsible for. The diagram in Figure B.3 shows the class descriptions that manage the plans inside cells.

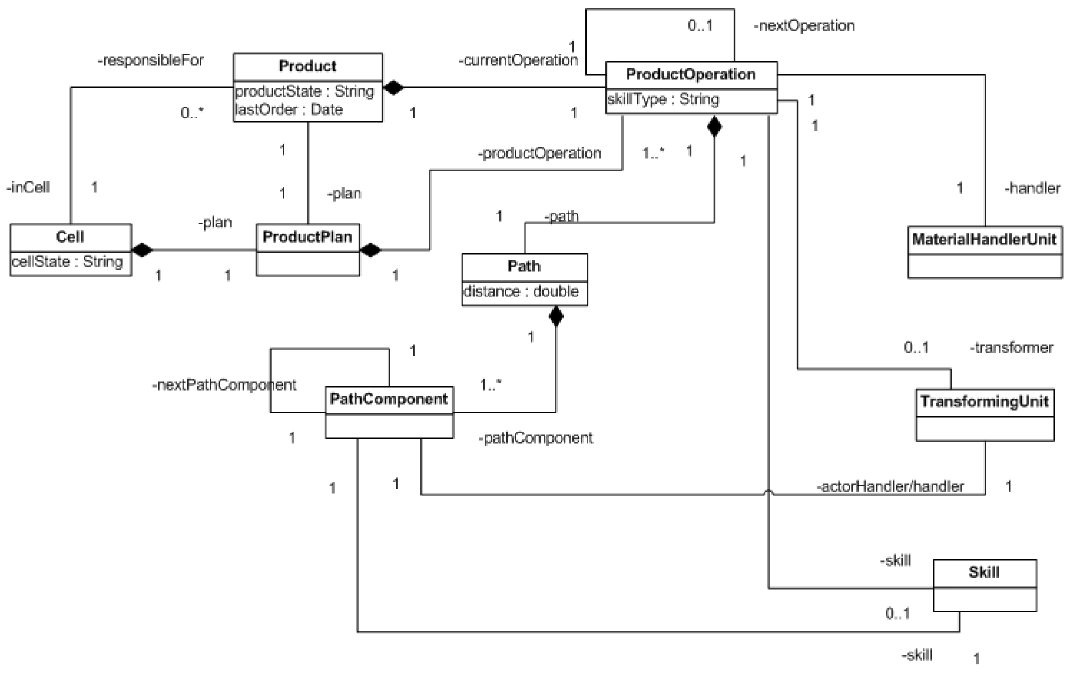


Figure B.3: UML class diagram of products and plans.

A cell contains a product plan, which states the operations to be performed in the product. Each operation may contain a reference to the transforming unit that will perform an operation on the product and the respective skill used for it. It also holds a reference to the current material handling unit that should contain the product at the time of the operation. Since there may be more than one operation, each one contains a reference to the next operation to be performed.

Operations contain paths that are used to transport the product to the location where they should be performed. Each path component needs a reference to the material handling unit currently containing the product, the skill needed to transport the product to the next handler and the actor responsible for performing that skill. Also it has a reference to the next path component.

For the cell to keep track of the current product status, each product has a reference of its assigned plan, the operation currently being executed on it and the current material handling unit currently holding it.

This section of the ontology is not vital for the overall production task, since it may directly be defined as a simple process. Nevertheless, it may be used as a bridge for path-finding algorithms in future implementations, contributing to other approaches that already use these concepts to achieve self-organization.

Appendix C

An Example Process

The work described in this document, even if it had a simple objective, had a fairly complex background and all of its contents do not easily fit in a thesis this size. This chapter aims to explain a little better how processes are created by presenting a simple practical example, which may shed some light on many doubts that may have arose during the reading of this document.

In this example, the process receives a String input containing a given message. It will, then, generate an unique identification and, afterwards, print a line of text in the console with the concatenation of the message with the generated identification. It may be executed inside an agent and, since it does not possess a starting condition, the agent must be explicitly requested to do so.

In order to implement this process, the following two activity types were needed, which were defined in external ontological descriptions and Java libraries in the way already explained in Section 6.3:

- **GenerateUniqueID** - This activity generates an unique identification based on the current time stamp and the hash code of the agent it is executed in. This identification is outputted as a String. This activity type was created specifically for this example but used a Java method already defined in the current framework.

- **Print** - Simply prints a line of text in the console. The text printed needs to be an input of this type of activity. This activity type was not created just for the current example and is used in many processes implemented in this thesis.

The process itself contains a single sequence flow control, which, in turn contains the two instances of the needed activities, as well as a connection between them. This way, the execution will flow from the activity that generates the identification to the activity that will print the text. Figure C.1 shows the main sequence control of the process, as created in the Process Editor.

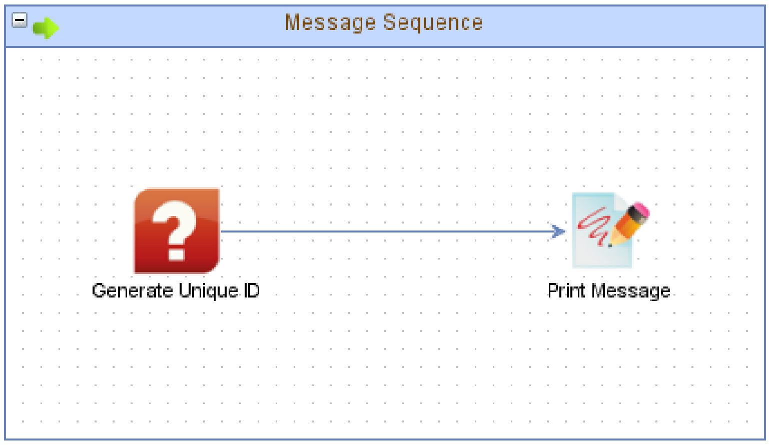


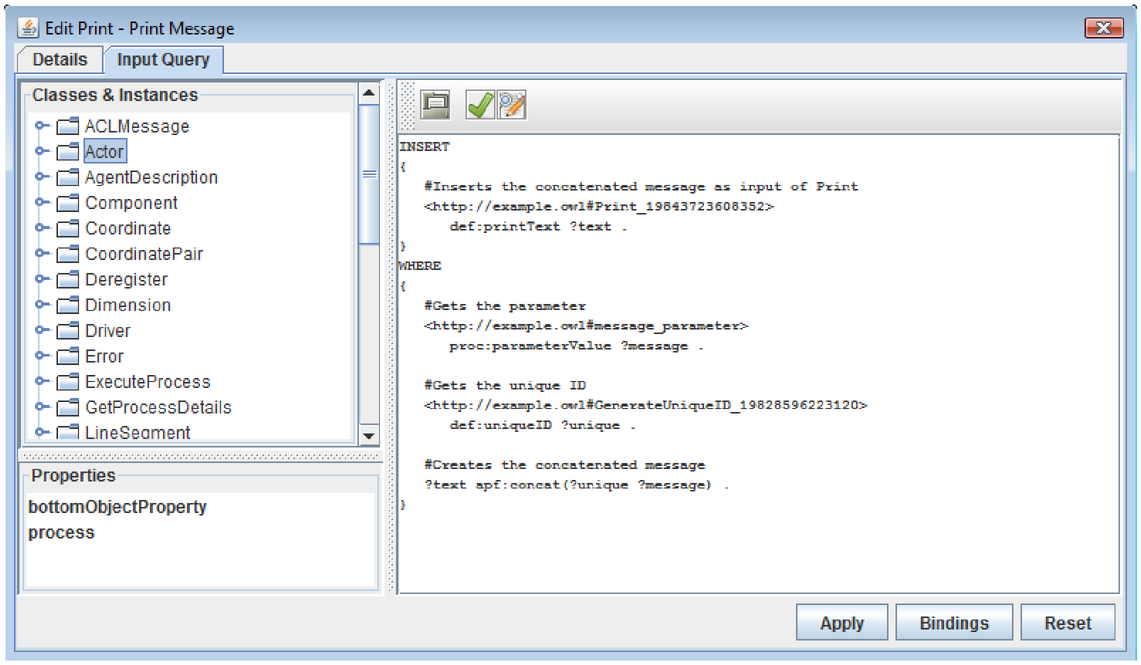
Figure C.1: A section of the example process, as seen in the editor.

It is very useful that instances, in processes or any other ontology, have an identification for easier querying. In this example, the URI identifications defined for the most important instances shown in Table C.1:

Table C.1: The set of instances inside the example process.

Instance	**Class**
http://example.owl#actor1	Actor
http://example.owl#example_process	Process
http://example.owl#GenerateUniqueID_19828596223120	GenerateUniqueID
http://example.owl#Print_19843723608352	Print
http://example.owl#message_parameter	Parameter

Activities to generate unique identifications do not require any inputs, but activities to print lines of text do. Figure C.2 shows how the input for the print activity was created. The SPARQL/Update query was inserted in the configuration of the print instance, as defined in the process model specification. Basically, the query states that the output of the previous activity and the value of the input parameter of the process are to be concatenated and the result of this operation is inserted as an input of the print activity, using the property *printText*.



```
INSERT
{
   #Inserts the concatenated message as input of Print
   <http://example.owl#Print_19843723608352>
      def:printText ?text .
}
WHERE
{
   #Gets the parameter
   <http://example.owl#message_parameter>
      proc:parameterValue ?message .

   #Gets the unique ID
   <http://example.owl#GenerateUniqueID_19828596223120>
      def:uniqueID ?unique .

   #Creates the concatenated message
   ?text apf:concat(?unique ?message) .
}
```

Figure C.2: The input of the print activity, as seen in the editor.

Another important subject is that the *concat* function is a pre-defined SPARQL function provided by Jena. This API has many more useful features for querying ontological KBs, which helped in many processes throughout this work. More information on these features may be accessed through the Jena website [Hewlett-Packard, 2010].

As a result of the execution of this process inside a deployed agent, if the input message is *"Hello, World!"*, the printed text will be a concatenation of this message with an unique identification, *"ID_17970505_3835313490176Hello, World!"*.

This is a very simple example with no practical use but it helps explaining how the processes can be created. Every agent of this thesis can be configured using this principle without the need for hard-coding.

To better understand this example, the full OWL code is displayed in Listing C.1. The listing shows a fairly large code for such a simple process. This is the reason why an editor was very useful for the creation of the presented system. A good examination of the code, may help clearing some doubts about the work developed in this thesis, tying some loose ends.

Listing C.1: The full OWL code of the presented example.

```
<rdf:RDF
    xmlns:def="http://www.fct.unl.pt/ontologies/eas-process-default-ontology.owl#"
    xmlns:proc="http://www.fct.unl.pt/ontologies/eas-process-ontology.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:agt="http://www.fct.unl.pt/ontologies/eas-agent-ontology.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns="http://www.fct.unl.pt/ontologies/default.owl#"
    xmlns:cnp="http://www.fct.unl.pt/ontologies/eas-concepts-ontology.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <owl:Ontology rdf:about="http://www.fct.unl.pt/ontologies/default.owl">
    <owl:imports rdf:resource="http://www.fct.unl.pt/ontologies/eas-process-eas-ontology.owl"/>
    <owl:imports rdf:resource="http://www.fct.unl.pt/ontologies/eas-process-default-ontology.owl"/>
    <owl:imports rdf:resource="http://www.fct.unl.pt/ontologies/eas-process-ontology.owl"/>
  </owl:Ontology>
  <proc:Actor rdf:about="http://example.owl#actor1">
    <proc:process>
      <proc:Process rdf:about="http://example.owl#example_process">
        <proc:hasStartControl>
          <proc:Sequence rdf:about="http://example.owl#Sequence_19806796742941">
            <proc:nodeDimension>
              <cnp:Dimension>
                <cnp:height rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                >240.0</cnp:height>
                <cnp:width rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                >420.0</cnp:width>
              </cnp:Dimension>
            </proc:nodeDimension>
            <proc:nodePosition>
              <cnp:Coordinate>
                <cnp:y rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                >140.0</cnp:y>
                <cnp:x rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                >240.0</cnp:x>
              </cnp:Coordinate>
            </proc:nodePosition>
            <proc:hasLastNode>
              <def:Print rdf:about="http://example.owl#Print_19843723608352">
                <proc:nodeDimension>
                  <cnp:Dimension>
                    <cnp:height rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >50.0</cnp:height>
                    <cnp:width rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >50.0</cnp:width>
                  </cnp:Dimension>
                </proc:nodeDimension>
                <proc:nodePosition>
                  <cnp:Coordinate>
                    <cnp:y rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >125.0</cnp:y>
                    <cnp:x rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >335.0</cnp:x>
                  </cnp:Coordinate>
                </proc:nodePosition>
                <proc:inputQuery>
                  <proc:SPARQLQuery>
                    <proc:queryText rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                    >PREFIX comm: &lt;http://www.fct.unl.pt/ontologies/eas-comm-ontology.owl#&gt;
PREFIX protege: &lt;http://protege.stanford.edu/plugins/owl/protege#&gt;
PREFIX xsp: &lt;http://www.owl-ontologies.com/2005/08/07/xsp.owl#&gt;
PREFIX cnp: &lt;http://www.fct.unl.pt/ontologies/eas-concepts-ontology.owl#&gt;
PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
PREFIX swrl: &lt;http://www.w3.org/2003/11/swrl#&gt;
PREFIX def: &lt;http://www.fct.unl.pt/ontologies/eas-process-default-ontology.owl#&gt;
PREFIX owl: &lt;http://www.w3.org/2002/07/owl#&gt;
PREFIX xsd: &lt;http://www.w3.org/2001/XMLSchema#&gt;
PREFIX swrlb: &lt;http://www.w3.org/2003/11/swrlb#&gt;
PREFIX rdf: &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#&gt;
PREFIX agt: &lt;http://www.fct.unl.pt/ontologies/eas-agent-ontology.owl#&gt;
PREFIX proc: &lt;http://www.fct.unl.pt/ontologies/eas-process-ontology.owl#&gt;
PREFIX apf: &lt;java:com.hp.hpl.jena.query.pfunction.library.&gt;
PREFIX list: &lt;http://jena.hpl.hp.com/ARQ/list#&gt;

INSERT
{
        #Inserts the concatenated message as input of Print
        &lt;http://example.owl#Print_19843723608352&gt;
                def:printText ?text .
}
WHERE
{
        #Gets the parameter
        &lt;http://example.owl#message_parameter&gt;
                proc:parameterValue ?message .

        #Gets the unique ID
        &lt;http://example.owl#GenerateUniqueID_19828596223120&gt;
                def:uniqueID ?unique .

        #Creates the concatenated message
        ?text apf:concat(?unique ?message) .
}</proc:queryText>
                  </proc:SPARQLQuery>
                </proc:inputQuery>
                <proc:description rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                ></proc:description>
                <proc:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                >Print Message</proc:name>
              </def:Print>
            </proc:hasLastNode>
            <proc:hasFirstNode>
              <def:GenerateUniqueID rdf:about="http://example.owl#GenerateUniqueID_19828596223120">
                <proc:nodeDimension>
                  <cnp:Dimension>
                    <cnp:height rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >50.0</cnp:height>
                    <cnp:width rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >50.0</cnp:width>
                  </cnp:Dimension>
                </proc:nodeDimension>
                <proc:nodePosition>
                  <cnp:Coordinate>
                    <cnp:y rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >125.0</cnp:y>
                    <cnp:x rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
                    >95.0</cnp:x>
                  </cnp:Coordinate>
                </proc:nodePosition>
                <proc:description rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                ></proc:description>
                <proc:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                >Generate Unique ID</proc:name>
              </def:GenerateUniqueID>
            </proc:hasFirstNode>
            <proc:hasTransition>
```

```
              <proc:Transition rdf:about="http://example.owl#Transition_19853776841362">
                <proc:to rdf:resource="http://example.owl#Print_19843723608352"/>
                <proc:from rdf:resource="http://example.owl#GenerateUniqueID_19828596223120"/>
              </proc:Transition>
            </proc:hasTransition>
            <proc:hasActivity rdf:resource="http://example.owl#Print_19843723608352"/>
            <proc:hasActivity rdf:resource="http://example.owl#GenerateUniqueID_19828596223120"/>
            <proc:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >Message Sequence</proc:name>
          </proc:Sequence>
        </proc:hasStartControl>
        <proc:hasInputParameter>
          <proc:Parameter rdf:about="http://example.owl#message_parameter">
            <proc:parameterName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >Message</proc:parameterName>
            <proc:paramType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
            >http://www.w3.org/2001/XMLSchema#string</proc:paramType>
          </proc:Parameter>
        </proc:hasInputParameter>
        <proc:actor rdf:resource="http://example.owl#actor1"/>
        <proc:enabled rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
        >true</proc:enabled>
      </proc:Process>
    </proc:process>
  </proc:Actor>
</rdf:RDF>
```

Bibliography

[Alsafi and Vyatkin, 2010] Alsafi, Y. and Vyatkin, V. (2010). Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing. *Robotics and Computer-Integrated Manufacturing*, 26(4):381–391.

[Barata, 2005] Barata, J. (2005). *Coalition Based Approach for Shop Floor Agility - A Multiagent Approach*. PhD thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

[Barata et al., 2005] Barata, J., Camarinha-Matos, L., and Onori, M. (2005). A multiagent based control approach for evolvable assembly systems. In *3rd IEEE Intemational Conference on Industrial Informatics (INDIN)*.

[Barata and Camarinha-Matos, 2002] Barata, J. and Camarinha-Matos, L. M. (2002). Shop floor re-engineering using agents. In *33rd International Symposium on Robotics*.

[Barata and Onori, 2006] Barata, J. and Onori, M. (2006). Evolvable assembly and exploiting emergent behaviour. In *2006 IEEE International Symposium on Industrial Electronics*, volume 4, pages 3353–3360.

[Barata et al., 2007] Barata, J., Onori, M., Frei, R., and Leitão, P. (2007). Evolvable production systems: Enabling research domains. In *Int. Conf. on Changeable, Agile, Reconfigurable and Virtual Production (CARV)*, pages 239–244, Toronto, Ontario, Canada.

[Bellifemine et al., 2010] Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2010). *Jade programmers guide*. TILAB.

[Bonitasoft, 2011] Bonitasoft (2011). Bonita open solution. http://www.bonitasoft.com/.

[Brooks, 1991a] Brooks, R. (1991a). Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159.

[Brooks, 1991b] Brooks, R. A. (1991b). Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence(IJCAI-91)*, pages 569–595, Sydney, Australia.

[Browne et al., 1988] Browne, J., Harhen, J., and Shivnan, J. (1988). *Production management systems: a CIM perspective*. Addison Wesley Publishing Company.

[Camarinha-Matos et al., 1995] Camarinha-Matos, L., Pinheiro-Pita, H., Rabelo, R., and Barata, J. (1995). Towards a taxonomy of cim activities. *International Journal of Computer Integrated Manufacturing*, 8(3):160–176.

[Colombo et al., 2006] Colombo, A. W., Schoop, R., and Neubert, R. (2006). An agent-based intelligent control platform for industrial holonic manufacturing systems. *IEEE Transactions on industrial electronics*, 53(1):322–327.

[Damásio, 2003] Damásio, A. (2003). *Ao Encontro de Espinosa: As Emoções Sociais e a Neurologia do Sentir*. Publicações Europa América.

[Davenport, 1993] Davenport, T. H. (1993). *Process innovation: reengineering work through information technology*. Ernst & Young.

[Dilts et al., 1991] Dilts, D. M., Boyd, N. P., and Whorms, H. H. (1991). The evolution of control architectures for automated manufacturing systems. *Journal of Manufacturing Systems*, 10(1):79–93.

[ElMaraghy, 2005] ElMaraghy, H. (2005). Flexible and reconfigurable manufacturing systems paradigms. *International journal of flexible manufacturing systems*, 17(4):261–276.

[EUPASS, 2006] EUPASS (2006). Evolvable ultraprecision assembly systems. http://www.eupass.org/.

[Ferguson, 1992] Ferguson, I. A. (1992). *Touring Machines: an Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, University of Cambridge, Cambridge.

[FIPA, 2002a] FIPA (2002a). Fipa acl message structure specification. http://www.fipa.org/specs/fipa00061/.

[FIPA, 2002b] FIPA (2002b). Fipa query interaction protocol specification. http://www.fipa.org/specs/fipa00027/.

[FIPA, 2002c] FIPA (2002c). Fipa request interaction protocol specification. http://www.fipa.org/specs/fipa00026/.

[FIPA, 2002d] FIPA (2002d). Fipa subscribe interaction protocol specification. http://www.fipa.org/specs/fipa00035/.

[FIPA, 2000] FIPA, T. (2000). Fipa communicative act library specification. http://www.fipa.org/specs/fipa00037/.

[Ford and Crowther, 1926] Ford, H. and Crowther, S. (1926). *Today and tomorrow*. Doubleday.

[Frei and Barata, 2008] Frei, R. and Barata, J. (2008). Embodied intelligence to turn evolvable assembly systems reality. *Innovation in Manufacturing Networks*, pages 269–278.

[Frei et al., 2010] Frei, R., Di Marzo Serugendo, G., Pereira, N., Belo, J., and Barata, J. (2010). Implementing self-organisation and self-management in evolvable assembly systems. In *Proc. IEEE Int Industrial Electronics (ISIE) Symp*, pages 3527–3532.

[Frei et al., 2009] Frei, R., Pereira, N., Belo, J., Barata, J., and Serugendo, G. (2009). Self-awareness in evolvable assembly systems. Technical report, Technical Report BBKCS-09-08, School of Computer Science and Information Systems, Birkbeck College, London, UK, 2009.

[Genesereth and Nilsson, 1987] Genesereth, M. R. and Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

[Gennari et al., 2003] Gennari, J. H., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubzy, M., Eriksson, H., Noy, N. F., and Tu, S. W. (2003). The evolution of protg: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 58(1):89 – 123.

[Georgakopoulos et al., 1995] Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153.

[Groover, 2007] Groover, M. (2007). *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall.

[Gross et al., 1996] Gross, D. et al. (1996). *Forbes greatest business stories of all time*. Wiley.

[Gruber et al., 1993] Gruber, T. et al. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5:199–199.

[Guarino and Poli, 1995] Guarino, N. and Poli, R. (1995). Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human Computer Studies*, 43(5):625–640.

[Guarino et al., 1993] Guarino, N., Poli, R., et al. (1993). Toward principles for the design of ontologies used for knowledge sharing. In *In Formal Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers, in press. Substantial revision of paper presented at the International Workshop on Formal Ontology*. Citeseer.

[Gullander, 1999] Gullander, P. (1999). On reference architectures for development of flexible cell control systems. *Doktorsavhandlingar vid Chalmers Tekniska Hogskola*, (1502):1–166.

[Hewlett-Packard, 2010] Hewlett-Packard (2010). Jena a semantic web framework for java. http://jena.sourceforge.net/.

[Hill et al., 2008] Hill, J., Pezzini, M., and Natis, Y. (2008). Findings: confusion remains regarding bpm terminologies. *Gartner Research, Stamford, CT*, 501(G00155817).

[Huff and Edwards, 1999] Huff, B. L. and Edwards, C. R. (1999). Layered supervisory control architecture for reconfigurable automation. *Production, Planning and Control*, 10(7):659–670.

[JADE, 2010] JADE (2010). Java agent development framework. http://jade.tilab.com/.

[Labrou et al., 1999] Labrou, Y., Finin, T., , and Peng, Y. (1999). Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 45(2):45–52.

[Laclavık et al., 2006] Laclavık, M., Balogh, Z., Babık, M., and Hluchỳ, L. (2006). Agentowl: Semantic knowledge model and agent architecture. *Computing and Informatics*, 25:419–437.

[Leitão and Restivo, 2006] Leitão, P. and Restivo, F. (2006). Adacor: A holonic architecture for agile and adaptive manufacturing control. *Computers in Industry*, 57(2):121 – 130.

[Leitao, 2004] Leitao, P. (2004). *An agile and adaptive holonic architecture for manufacturing control.* PhD thesis, Faculty of Engineering, University of Porto, Portugal.

[Lemaignan et al., 2006] Lemaignan, S., Siadat, A., Dantan, J.-Y., and Semenenko, A. (2006). Mason: A proposal for an ontology of manufacturing domain. In *Proc. IEEE Workshop Distributed Intelligent Systems: Collective Intelligence and Its Applications DIS 2006*, pages 195–200.

[Lin et al., 2011] Lin, L., Zhang, W., Lou, Y., Chu, C., and Cai, M. (2011). Developing manufacturing ontologies for knowledge reuse in distributed manufacturing environment. *International Journal of Production Research*, 49(2):343–359.

[Maraldo et al., 2006] Maraldo, T., Onori, M., Barata, J., and Semere, D. (2006). Evolvable assembly systems: Clarifications and developments to date. *CIRP / IWES 6th International Workshop on Emergent Synthesis.*

[Maskell, 2001] Maskell, B. (2001). The age of agile manufacturing. *Supply Chain Management: An International Journal*, 6(1):5–11.

[Mercian et al., 2006] Mercian, M., Korclic, V., Zoitl, A., and Lazinica, A. (2006). Knowledge-based multi-agent architecture. In *Computational Intelligence for Modelling, Control and Automation, 2006 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, pages 55 –55.

[Merdan et al., 2008] Merdan, M., Koppensteiner, G., Hegny, I., and Favre-Bulle, B. (2008). Application of an ontology in a transport domain. In *Industrial Technology, 2008. ICIT 2008. IEEE International Conference on*, pages 1–6.

[Microsoft, 2011] Microsoft (2011). Microsoft biztalk. http://www.microsoft.com/biztalk/en/us/overview.aspx.

[Miller and Walker, 1990] Miller, R. and Walker, T. (1990). *FMS/CIM systems integration handbook.* Fairmont Press.

[Muller and Pischel, 1993] Muller, J. P. and Pischel, M. (1993). *The Agent Architecture InteRRaP: Concept and Application*. DFKI Saarbrucken.

[Noy et al., 2001] Noy, N., McGuinness, D., et al. (2001). Ontology development 101: A guide to creating your first ontology.

[Onori et al., 2005] Onori, M., Alsterman, H., and Barata, J. (2005). An architecture development approach for evolvable assembly systems. In *Assembly and Task Planning: From Nano to Macro Assembly and Manufacturing, 2005. (ISATP 2005). The 6th IEEE International Symposium on*, pages 19 –24.

[Onori et al., 2006] Onori, M., Barata, J., and Frei, R. (2006). Evolvable assembly systems basic principles. *Information Technology for Balanced Manufacturing Systems*, pages 317–328.

[Pereira, 2011] Pereira, N. (2011). An agent-based evolutionary approach for manufacturing system layout design. Unpublished Masters Thesis.

[Pine and Davis, 1999] Pine, B. and Davis, S. (1999). *Mass customization: The new frontier in business competition*. Harvard Business School Pr.

[Piore and Sabel, 1984] Piore, M. and Sabel, C. (1984). *The second industrial divide: possibilities for prosperity*. Basic books.

[Pouchard et al., 2000] Pouchard, L., Ivezic, N., and Schlenoff, C. (2000). Ontology engineering for distributed collaboration in manufacturing. In *Proceedings of the AIS2000 Conference*. Citeseer.

[R. Nagel, 1992] R. Nagel, R. D. (1992). 21st century manufacturing enterprise strategy. Technical report, Iacocca Institute, Lehigh University, USA.

[Ranky, 1990] Ranky, P. (1990). *Flexible manufacturing cells and systems in CIM*. CIMWare.

[Rembold et al., 1993] Rembold, U., Nnaji, B., and Storr, A. (1993). *Computer integrated manufacturing and engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Ribeiro et al., 2008] Ribeiro, L., Barata, J., and Colombo, A. (2008). Mas and soa: A case study exploring principles and technologies to support self-properties in assembly systems. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on*, 0:192–197.

[Ryan K.L. Ko, 2009] Ryan K.L. Ko, Stephen S.G. Lee, E. W. L. (2009). Business process management (bpm) standards: a survey. *Business Process Management Journal*, 15(5):744 – 791.

[Shen et al., 2006] Shen, W., Onori, M., Barata, J., and Frei, R. (2006). Evolvable assembly systems basic principles. In *Information Technology For Balanced Manufacturing Systems*, volume 220 of *IFIP International Federation for Information Processing*, pages 317–328. Springer Boston.

[Shivanand, 2006] Shivanand, H. (2006). *Flexible Manufacturing System*. New Age International.

[Sirin and Parsia, 2007] Sirin, E. and Parsia, B. (2007). Sparql-dl: Sparql query for owl-dl. In *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, volume 4. Citeseer.

[Smith, 2009] Smith, B. (2009). Ontology and information systems.

[Sudarsan et al., 2005] Sudarsan, R., Fenves, S., Sriram, R., and Wang, F. (2005). A product information modeling framework for product lifecycle management. *Computer-Aided Design*, 37(13):1399–1411.

[TIBCO, 2011a] TIBCO (2011a). Tibco activematrix bpm. http://www.tibco.com/products/bpm/default.jsp.

[TIBCO, 2011b] TIBCO (2011b). Tibco activematrix businessworks. http://www.tibco.com/products/soa/composite-applications/activematrix-businessworks/.

[Upton, 1990] Upton, D. (1990). Flexible structure for computer controlled manufacturing system. *Manufacturing Review*, 5(1):5874.

[van der Aalst et al., 2003] van der Aalst, W., Hofstede, A. H. M. T., and Weske, M. (2003). Business process management: A survey. In *Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS*, pages 1–12. Springer-Verlag.

[Victor and Boynton, 1998] Victor, B. and Boynton, A. (1998). *Invented here: Maximizing your organization's internal growth and profitability*. Harvard Business Press.

[Vos, 2001] Vos, J. (2001). *Module and System Design in Flexible Automated Assembly*. doctoral thesis, TU Delft, Delft University of Technology.

[W3C, 2004a] W3C (2004a). Owl-s: Semantic markup for web services. http://www.w3.org/Submission/OWL-S/.

[W3C, 2004b] W3C (2004b). Owl web ontology language overview. http://www.w3.org/TR/owl-features/.

[W3C, 2004c] W3C (2004c). Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/rdf-schema/.

[W3C, 2004d] W3C (2004d). Rdf/xml syntax specification. http://www.w3.org/TR/REC-rdf-syntax/.

[W3C, 2008a] W3C (2008a). Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/.

[W3C, 2008b] W3C (2008b). Sparql update. http://www.w3.org/Submission/SPARQL-Update/.

[WADE, 2010] WADE (2010). Wade - workflow agent development environment. http://jade.tilab.com/wade/.

[WfMC, 2008] WfMC (2008). Xpdl. http://www.wfmc.org/xpdl.html.

[Womack et al., 1990] Womack, J., Jones, D., and Roos, D. (1990). *The machine that changed the world*.

[Wooldridge, 2000] Wooldridge, M. (2000). *Reasoning about Rational Agents*. MIT Press, Cambridge, MA.

[Wooldridge, 2002] Wooldridge, M. (2002). *Introduction to MultiAgent Systems*. John Wiley & Sons.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. (1995). Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152.

[Wooldridge and Jennings, 1994] Wooldridge, M. and Jennings, N. R. (1994). Agent theories, architectures and languages: A survey. In *ECAI-Workshop on Agent Theories, Architectures and Languages*.